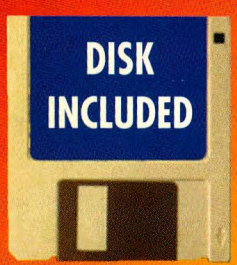
THE WAITE GROUP®

GARDENS OF IMAGINATION

Programming 3D Maze Games in C/C++

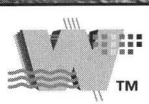


INCLUDES SAMPLE GAME, DESIGN UTILITIES, SOURCE CODE, AND MORE!

CHRISTOPHER LAMPTON

Gardens of Imagination

Christopher Lampton



Waite Group Press™
Corte Madera, California

Publisher • Mitchell Waite
Editor-in-Chief • Scott Calamar
Editorial Director • Joel Fugazzotto
Managing Editor • John Crudo
Content Editor • Harry Henderson
Technical Reviewers • Mike Radtke and David Bolman
Production Director • Julianne Ososke
Design • Michele Cuneo
Project Coordinator • Kristin Peterson
Production • William Salit • the BOOKWORKS
Illustrations • Ben Long
Cover Design • Ted Mader & Associates

© 1994 by The Waite Group, Inc.®
Published by Waite Group Press™, 200 Tamal Plaza, Corte Madera, CA 94925.

Waite Group Press is distributed to bookstores and book wholesalers by Publishers Group West, Box 8843, Emeryville, CA 94662, 1-800-788-3123 (in California 1-510-658-3453).

All rights reserved. No part of this manual shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, desktop publishing, recording, or otherwise, without permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

All terms mentioned in this book that are known to be registered trademarks, trademarks, or service marks are listed below. In addition, terms suspected of being trademarks, registered trademarks, or service marks have been appropriately capitalized. Waite Group Press cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any registered trademark, trademark, or service mark.

The Waite Group is a registered trademark of The Waite Group, Inc.
Waite Group Press and The Waite Group logo are trademarks of The Waite Group, Inc.
Borland C++ is a registered trademark of Borland International, Inc.
All other product names are trademarks, registered trademarks, or service marks of their respective owners.

Printed in the United States of America
94 95 96 97 • 10 9 8 7 6 5 4 3 2 1

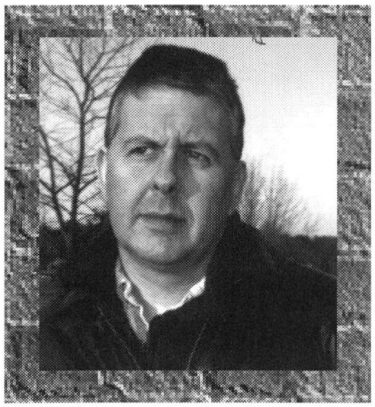
Library of Congress Cataloging-in-Publication Data

Lampton, Christopher
Gardens of imagination : programming 3D maze games in Borland C++
/ Christoper Lampton.
p. cm.
Includes bibliographical references and index.
ISBN: 1-878739-59-X: $34.95
1. Computer games. 2. Borland C++. I. Title
GV1469.2.L36 1994 94-7917
794.8'15365--dc30 CIP

Dedication

For Susan Wiener, without whom this
and quite a few other books could never have been written.

And for Lyric, in memoriam.



About the Author

Christopher Lampton is the author of more than 80 books for readers young and old. These include 20 books on microcomputers and computer programming, including introductory books on BASIC, Pascal, and assembly language programming, and four books on computer graphics and animation programming. He has also written books on topics as diverse as biotechnology, airline safety, underwater archaeology, sound, astronomy, dinosaurs, the origin of the universe, and predicting the course of epidemics. He holds a degree in broadcast communications from the University of Maryland, College Park, and has worked both as a disk jockey and as a producer of television commercials for a Maryland TV station. When he is not writing, programming, or indulging his hobby as a fanatic computer gamer, he serves as Associate Sysop (system operator) of the Game Publishers Forums (GAMPUB) on the CompuServe Information Service. He is also the author of Waite Group Press' *Flights of Fantasy* and *Nanotechnology Playhouse.*

Table of Contents

INTRODUCTION xvii

INSTALLATION xix

CHAPTER ONE Lost in the Maze. 1

CHAPTER TWO Basic Graphic Techniques 15

CHAPTER THREE Basic Input Techniques 61

CHAPTER FOUR Manipulating Bitmaps 105

CHAPTER FIVE Polygon Mazes 161

CHAPTER SIX Texture Mapping. 193

CHAPTER SEVEN Space: The Final Frontier . . . 227

CHAPTER EIGHT Ray Tracing 259

CHAPTER NINE Ray Casting 293

CHAPTER TEN Heightmapping 357

CHAPTER ELEVEN Lightsourcing 399

CHAPTER TWELVE Optimization 459

CHAPTER THIRTEEN Putting It All Together 521

APPENDIX A 551

APPENDIX B 563

INDEX 573

Contents

INTRODUCTION XVII

INSTALLATION XIX

CHAPTER ONE Lost in the Maze 1

The Once-and-Future Maze Game 4
- Face to Face with (Virtual) Reality 7
- A Maze in Wolf's Clothing 9

Types of Mazes 12
- Wireframe Mazes 12
- Bitmapped Mazes 13
- Polygon Mazes 13
- Ray-Cast Mazes 13

Animation Programming 14

CHAPTER TWO Basic Graphic Techniques 15

Video Memory: A Brief Tour 18
- Offsets and Addresses 18
- Manipulating Memory with Pointers 19
- Mode 13h 21
- Setting the Video Mode 22
- Finding Video Memory 24

The VGA Palette 27

A Screen-Clearing Function 30

Building a Wireframe Maze 32
- The Cartesian Coordinate System 32
- Plotting Pixels 33
- Drawing a Line 35
- Endpoints 36
- Horizontal and Vertical Lines 37
- Lines of Arbitrary Slope 40
- Bresenham's Algorithm 43
- At Last ... the Maze 46
- One Square at a Time 48

CHAPTER THREE Basic Input Techniques 61

The PC Keyboard 63
Two Keys at a Time 64
The Keyboard Interrupt 65
Swiping the Interrupt 66
The initkey() Function 67
The remkey() Function 69
The newkey() Function 69
Using the Keyboard Functions 73
The SCANKEY Program 74
Removing the Keyboard Handler 76
The PC Mouse 76
The Mouse Driver 77
The initmouse() Function 79
The readmbutton() Function 79
The relpos() Function 80
Using the Mouse Functions 81
The MOUSE Program 82
The PC Joystick 84
Analog Joysticks 84
Reading the Joystick Buttons 85
Determining the Stick Position 86
Using the Joystick Functions 88
Calibrating the Joystick 89
The JOYSTICK Program 89
The Event Manager 91
Maze Events 93
The init_events() Function 93
The Joystick Calibration Functions 93
The getevent() Function 95
Animating the Maze 97
Fixing the Flicker 100
The GOMAZE Program 101

CHAPTER FOUR Manipulating Bitmaps 105

What Is a Bitmap? 109
Graphics File Formats 109
The 8-Bit Targa Format 110

An 8-Bit Targa Loader 112
The loadTGA() Function 115
An 8-Bit Targa Display Program 117
The TGASHOW Program 119
The PCX File Format 121
Run-Length Encoding 122
The loadPCX() Function 124
The load_image() Function 125
The load_palette() Function 127
The PCXSHOW Program 128
Building a Bitmapped Maze 129
Ray Tracing the Maze 130
Nine Views of the Maze 131
Slicing and Dicing the Maze 134
The Maze-Drawing Algorithm 134
The Bitmapped drawmaze() Function 140
The draw_slice() Function 149
The bitmaze() Program 149
Touring the Maze 157
Refining the Maze Engine 158
Flying Through the Maze 160

CHAPTER FIVE Polygon Mazes 161

Filled and Unfilled Polygons 164
Filling a Polygon 167
The polytype Structure 169
Polygon Clipping 169
A Polygon-Drawing Function 171
The Polygon-Drawing Algorithm 171
Clipping Against the Left Edge of the Viewport 173
Calculating the Slope 174
The Point-Slope Equation 175
The Drawing Begins 175
Clipping Against the Right Edge 176
The Error Terms 177
The Main Loop 177
Drawing the Line 178
The Final Details 178
The polydraw() Function 180
The POLYDEMO Program 182

The POLYCLIP Program....184
A Polygon Maze....185
A New Maze-Drawing Function....186
The New drawmaze() Function....188
Touring the Polygon Maze....190

CHAPTER SIX Texture Mapping....193

Two-and-a-Half-Dimensional Texture Mapping....196
The Two-Step Program....196
Bitmap Scaling....197
The Vertical Error Term....199
The Horizontal Error Term....200
Drawing a Pixel....201
The BITSCALE.CPP Program....202
Running the Program....206
Texture Mapping....206
The Ground Rules....207
The Texture-Mapped Polygon Function....208
The polytext() Function....212
A Textured Polygon Demo....215
The TEXTDEMO Program....215
Running the Program....217
A Texture-Mapped Maze....218
Running the Program....226
Going Three-Dimensional....226

CHAPTER SEVEN Space: The Final Frontier....227

The Third Dimension....231
Three-Dimensional Cartesian Space....232
The Three-Dimensional Illusion....233
Putting Perspective in Perspective....234
Two-and-a-Half Dimensions?....238
Direction Vectors....241
Vector Magnitude....244
Polar Coordinates....247
Polar Direction Vectors....250
The Unit Vector....253
The Unit Circle....254
Rotating a Point....254

Coordinate Transformations 255
Translating a Point 256
Rotating a Point 256
Three-Dimensional Rotation 257
Let 'Er Rip 258

CHAPTER EIGHT Ray Tracing 259

Soaking Up Rays 262
Looking at the World 262
A Ray Tracer of Our Own 264
Low-Resolution Rays 268
Defining the World 269
A-Tracing We Will Go 272
Follow That Ray 272
The Heart of the Ray Tracer 276
A Ray-Tracing Function 280
Tracing a Sphere 282
The lightsource() Function 284
The plot() Function 285
The RTDEMO Program 285
Extending the Ray Tracer 291
Ray-Traced Mazes? 291

CHAPTER NINE Ray Casting 293

Reducing the Number of Comparisons 296
Reducing the Number of Rays 296
Casting Rays 297
Follow That Ray 298
Wallcasting 298
The Fine Coordinate System 300
Rays Within the Grid 302
The Basic Ray-Casting Algorithm 303
Rays in All Directions 305
The Wallcasting drawmaze() Function 306
Stepping Through the Columns 308
A Touch of Trig 308
Rotating the Ray 309
Casting the Ray 311
The Slope of the Ray 312

Casting in a Loop 313
Calculating the Point of Intersection 313
Drawing the Wall 315
The drawmaze() Function 318
The WALLDEMO.CPP Program 321
Running WALLDEMO.EXE 324
Ray Casting and Texture Mapping 326
Finding the Bitmap Column 328
Clipping the Column 328
The Error Term Revisited 330
The Texture-Mapped Ray-Cast drawmaze() Function 332
Floorcasting 337
The Similar Triangles Method 340
Drawing the Ceiling 343
The Floorcasting drawmaze() Function 344
A Complete Ray-Casting Engine 347
The Floorcasting, Wallcasting draw_maze() Function 348

CHAPTER TEN Heightmapping 357

Block-Aligned Heightmaps 360
The floorheight[] Array 361
The wall[] and flor[] Arrays 361
Heightcasting 363
How High the Viewer? 364
Changes in Floor Height 365
The Block-Aligned draw_maze() Function 371
The BLOKDEMO Program 378
Heightmapped Ceilings 381
Tiled Heightmaps 382
Creating Height Tiles 383
The Tiled floorheight[] Array 384
The floorbase[] Array 384
The Tiled flor[] and wall[] Arrays 385
A Tiled Heightmapping Engine 386
The Tiled Heightmapped draw_maze() Function 389
The TILEDEMO.CPP Program 392
Hybrid Heightmapping Methods 396
Animated Heightmaps 397

CHAPTER ELEVEN Lightsourcing 399
Casting Some Light on Light 402
The Inverse Square Law 404
The Lightsourcing Formula 405
The MAKELITE.CPP Program 414
Using the Lightsourcing Tables 417
The LITEDEMO.CPP Program 419
Lightsourcing in Three Lines or Less 422
The Lightsourced draw_maze() Function 423
Alternate Lightsourcing Effects 430
Lightmapping 432
The Lightmapped draw_maze() Function 434
The MAPDEMO.CPP Program 441
Tiled Lightmaps 445
The Tiled Lightmapping draw_maze() Function 446
The TLMDEMO.CPP Program 453
Using Lightmaps 458

CHAPTER TWELVE Optimization 459
The Target Machine 462
A Few Optimization Tricks 462
Translating into Assembly Language 462
Unrolling the loop 463
Table Look-Ups 464
Fixed Point Arithmetic 465
Where to Optimize? 466
More About Fixed Point Math 467
Follow That Decimal! 470
Shifting Left and Right 471
Arithmetic Shifts 472
Double Precision Shifts 472
A Fixed Point Multiplication Function 473
A Fixed Point Division Function 475
Fixed Point Trigonometry 476
THE MAKETRIG.CPP Module 481
Optimized Wall Drawing 484
The drawwall() Function 488
Optimized Floors and Ceilings 489

The Optimized draw_maze() Function 500
The Optdemo Program 508
Running OPTDEMO.EXE 517
Optimization Review 517

CHAPTER THIRTEEN Putting It All Together 521

Adding Objects to the World 524
Hidden Surface Removal 524
Column by Column Drawing 527
The distance[] Array 527
Representing Objects 528
Drawing the Objects 531
The Painter's Algorithm 535
The drawobject() Function 537
Adding Game Features 539
Bells and Whistles 541
The Opening Credits 541
The Automap 546
Tracking the Player 548
Playing the Game 549

APPENDIX A Creating Game Graphics 551

POVRAY 554
Building a Maze 557

APPENDIX B Assembly Language 101 563

Assembly Language Pointers 566
Getting it into a Register 568
Assembly Language Odd and Ends 569
Passing Parameters to Assembly Language Procedures 570

INDEX 573

Acknowledgments

A book of this size and complexity is rarely the work of a single individual, even if only one name appears next to the title. More people than I can recall lent a helping hand on this one. Here's a partial list of those to whom I owe a heartfelt thanks:

Mitch Waite, my publisher, for letting me tackle another book in the vein of my earlier *Flights of Fantasy;* John Crudo, my editor, for retaining a sense of humor through crisis after crisis; Scott Calamar, The Waite Group's editorial director, for offering a friendly voice when it began to appear that this book would never be done; Mark Betz, for generously allowing me to reuse some of the terrific code that he wrote for *Flights of Fantasy;* and, of course, the denizens of the Game Design section of the CompuServe Gamers Forum, who are always willing to talk about cutting-edge game programming techniques. (You can get there yourself by logging onto the CompuServe Information Service and typing GO GAMERS. Look for game design discussions in section 11.)

Although I created many of the graphics images in this book myself, with the aid of the excellent public domain ray tracer POVRAY and the wonderful commercial art program Fractal Design Painter 2.0., I received additional (and invaluable) graphic assistance from two people: Judith Weller designed several of the graphics tiles used in the chapters on raycasting, including the blue stone wall, the skull door, and the floating skull, using 3D Studio Release 3 from Autodesk Software. Anyone interested in employing Judy's 3D modeling skills can contact her on CompuServe at ID 72662,536. Graphics designer Steven C. Blackmon, of the Digital Image Works, created the model of The Waite Group logo used in chapter 13. The image was modeled on and rendered by Imagine, a topnotch (and remarkably inexpensive) 3D modeler from Impulse Software. Steven can be contacted at CompuServe ID 73774,2504.

Finally, I owe special thanks to two people. Game designer Kevin Gliner helped me throughout this project, letting me read the code from his own three-dimensional raycasting system (and spending quite a few hours helping me to understand what I'd read). Several pieces of Kevin's code have been appropriated for use in this project, with his permission, as described in chapter 12. I've altered the code considerably to fit my needs, however, and Kevin should not be blamed for any bugs or clunky code that I've introduced in the process. It's only fair to mention that Kevin's raycasting engine runs considerably faster than the example

engine in this book and contains many clever optimizations beyond the ones discussed here. Anyone interested in licensing Kevin's engine can contact him at CompuServe ID 70363,3672.

And I'd like to offer thanks once again to Susan Wiener, who helped me keep body and soul together during this project. This book is dedicated explicitly to her, and to her recently departed (and much beloved) canary Lyric, but every other book I've written is dedicated to her in spirit.

Dear Reader:

What is a book? Is it perpetually fated to be inky words on a paper page? Or can a book simply be something that inspires—feeding your head with ideas and creativity regardless of the medium? The latter, I believe. That's why I'm always pushing our books to a higher plane; using new technology to reinvent the medium.

I wrote my first book in 1973, *Projects in Sights, Sounds, and Sensations.* I like to think of it as our first multimedia book. In the years since then, I've learned that people want to experience information, not just passively absorb it—they want interactive MTV in a book. With this in mind, I started my own publishing company and published *Master C,* a book/disk package that turned the PC into a C language instructor. Then we branched out to computer graphics with *Fractal Creations,* which included a color poster, 3-D glasses, and a totally rad fractal generator. Ever since, we've included disks and other goodies with most of our books. *Virtual Reality Creations* is bundled with 3-D Fresnel viewing goggles and *Walkthroughs and Flybys CD* comes with a multimedia CD-ROM. We've made complex multimedia accessible for any PC user with *Ray Tracing Creations, Multimedia Creations, Making Movies on Your PC, Image Lab,* and three books on Fractals.

The Waite Group continues to publish innovative multimedia books on cutting-edge topics, and of course the programming books that make up our heritage. Being a programmer myself, I appreciate clear guidance through a tricky OS, so our books come bundled with disks and CDs loaded with code, utilities, and custom controls.

By 1994, The Waite Group will have published 135 books. Our next step is to develop a new type of book, an interactive, multimedia experience involving the reader on many levels. With this new book, you'll be trained by a computer-based instructor with infinite patience, run a simulation to visualize the topic, play a game that shows you different aspects of the subject, interact with others on-line, and have instant access to a large database on the subject. For traditionalists, there will be a full-color, paper-based book.

In the meantime, they've wired the White House for hi-tech; the information super highway has been proposed; and computers, communication, entertainment, and information are becoming inseparable. To travel in this Digital Age you'll need guidebooks. The Waite Group offers such guidance for the most important software—your mind.

We hope you enjoy this book. For a color catalog, just fill out and send in the Reader Report Card at the back of the book.

Mitchell Waite
Publisher

Introduction

This book is a companion volume of sorts to my earlier Waite Group book, *Flights of Fantasy.* Where *Flights of Fantasy* concerned three-dimensional flight simulator programming, *Gardens of Imagination* tackles a subject that's a bit more down to earth — sometimes even *under* the earth.

Throughout this book, I refer to the type of game that it's about as "three-dimensional maze games," despite the fact that these games really have no collective name. The maze game encompasses a number of traditional computer game genres, including CRPGs, arcade games, and adventure games. What these games have in common is that each uses three-dimensional graphic techniques to depict the interior of a maze, which may in turn represent the interior of a dungeon, a spaceship, a jungle, even a castle occupied by Nazi stormtroopers. If you've played Ultima Underworld from Origin Systems, Wolfenstein 3D from Apogee Software, Doom from ID Software, Arena from Bethesda Softworks, or any of a host of other maze games to appear over the last few years, you should immediately recognize the type of game that I'm talking about. If not, you have a treat in store.

As in *Flights of Fantasy,* I've stocked this book from cover to cover with computer code demonstrating how to duplicate the effects in these games, including a working computer game developed in the last chapter. This code is intended not as a library of routines for the reader to plug into his or her programs (though it's possible to use the code as such), but as a series of examples to point the reader in the right direction and inspire him or her to produce code that lies even closer to the bleeding edge. Until recently, there was surprisingly little information available in print for programmers interested in writing commercial quality computer games. I hope that this book and *Flights of Fantasy* have helped to alleviate that situation at least a little.

This book includes a disk containing all the code that appears in these pages. It's probable that changes will be made to that code between the typesetting of the book and the mastering of the disk. If so, these changes will be documented in a file on the disk entitled README.TXT. If you have any problem getting this code to do what you want it to, or just want to tell me what you think of the book, you can contact me on CompuServe at ID 76711,301. Or log onto the Game Design Section of the CompuServe Gamers Forum (GO GAMERS).

Installation

Gardens of Imagination includes a disk that contains the source code, utilities, and maze discussed in the book. The files are in a compressed format, and should be installed on a hard drive before you use them; you'll need a minimum of 3MB of free hard disk space.

To install the files, follow these steps:

1. Change to root directory of the hard drive on which you wish to install the files. If the hard drive you wish to install the files on is the C: drive, at the DOS prompt you would type:

```
c:          ENTER
cd\         ENTER
```

If the hard drive is not C:, type the appropriate letter instead.

2. Create a directory for the files on the hard drive called GARDENS by typing:

```
md gardens      ENTER
```

You may use a directory name other than gardens, if you so prefer.

3. Change into the newly created directory. To change into the directory GARDENS, type:

```
cd gardens      ENTER
```

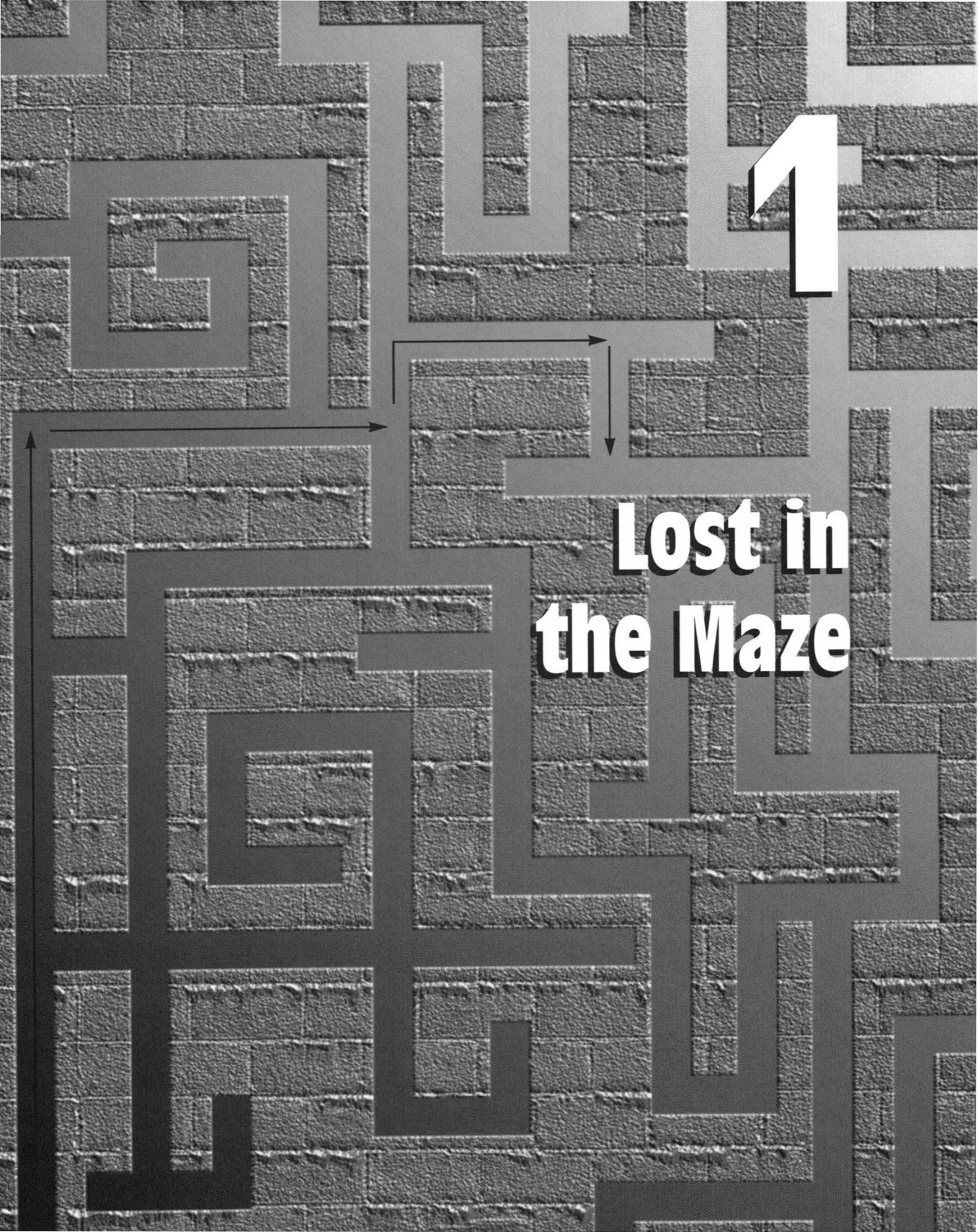
4. Insert the Gardens of Imagination disk into your 3.5-inch floppy drive. If that floppy drive is the A: drive, type:

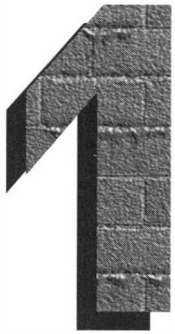
```
a:\gardens -d   ENTER
```

If it's the B: drive, type

```
b:\gardens -d   ENTER
```

Be sure to include the -d switch after gardens, so that the subdirectories and their files extract properly. Depending on the speed of your PC, the decompression and transfer of the Gardens of Imagination files may take a couple of minutes.


1
Lost in
the Maze



ou are trapped in a vast maze.

It's not that you're a pessimistic kind of person — you always like to look on the bright side of things, even when you need a flashlight to find it — but things are getting pretty grim. You ran out of food half an hour ago. You barely have enough magic left to heal a wart. Your hit points are so low that you can count them on your fingers and toes and still have enough digits left over to play "This Little Piggy."

To make matters worse, there's something out there in the darkness, something that's hungry and in a real bad mood. You hold up your flickering torch and try to get a look at it, but the flame is so low that you can barely see ten feet in front of you. Of course, that's probably for the best. If you can't see what's out there, maybe it can't see you either.

Some fun, huh?

Things could be worse. You had the foresight to save your game back in the last town, so even if that thing you hear licking its lips out there in the shadows decides to have you for dinner, you'll only lose 30 minutes of dungeon crawling. And maybe next time you'd better crawl in a different dungeon, until you have as much experience as some of those hypermuscular barbarians you've seen hanging around the adventurer's guild in recent weeks.

This dungeon is way too tough!

The Once-and-Future Maze Game

Fighting slavering monsters in a dark, maze-like dungeon may not be anybody's idea of fun, but when the dungeon is on the video display of a computer and the monsters are only bits and bytes shuttling through the silicon circuitry of a microprocessor, it can be a blast. The sort of computer role-playing game (or CRPG) that we've been describing in the last few paragraphs belongs to a larger genre of computer game, which we're going to refer to in this book as the maze game. CRPGs, however, are only one type of maze game. Other maze games would be better described as arcade games. And still others fall into the genre known as adventure games. All of these games, though, have one thing in common: They drop the player into the middle of a maze-like environment, and they depict that environment on the video display of the computer using 3D graphics. At a time when the buzzword "virtual reality" is on every game programmer's lips, these games put the player right in the thick of a virtual world.

If you like computer games, you've probably played more than a few that match this description. In this book, however, we're going to go beyond telling you how to play such games. We're going to tell you how to create them. If there's anything more fun than hacking and slashing your way through a virtual reality, it's creating your own virtual world!

Maze games are not new. They've been around almost as long as there have been microcomputers — and maybe a bit longer. The original Crowther and Woods adventure game, which first appeared on mainframes and minicomputers in the late 1960s, featured a sequence in which the player was trapped in a "maze of twisty little passages, all alike," though the maze was only described in words, not illustrated with graphics.

By the late 1970s, however, crude micromputer maze games had begun to appear. Some of these games rendered the interiors of mazes using simple line drawings, in which passageways were depicted by parallel lines extending away from the viewer toward the vanishing point, like something from the sketchbook of a Renaissance artist who had just discovered the laws of perspective. We'll refer to this type of depiction as a wireframe maze, by analogy to the wireframe graphics used in early microcomputer flight simulators. As crude as the graphics might have been, some of these early maze games were quite entertaining — and a couple of them spawned lines of sequels that are still appearing today.

Perhaps the most successful of these early maze games was Ultima, published in 1980 by then teenaged programmer Richard Garriott for the Apple II. Though most of the game took place on a scrolling two-dimensional map, the intrepid player could enter a series of dungeons scattered around the landscape and feast his or her eyes on state-of-the-art 1980 wireframe maze graphics. It wasn't necessary to enter the dungeons to win the game (though in later Ultima

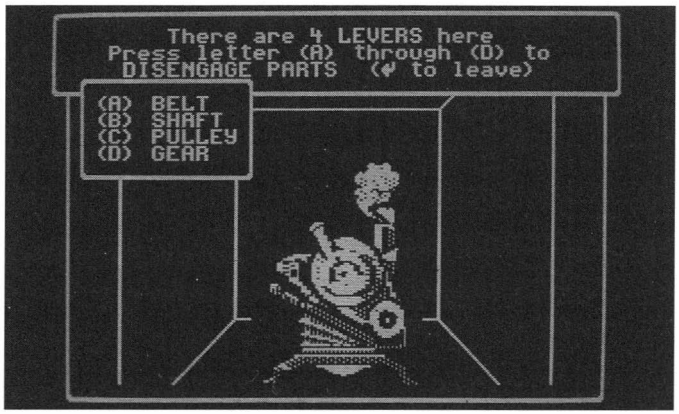


Figure 1-1 A scene from one of the early Wizardry games. The graphics may have been simple, but the games were big, complex, and quite popular

games it would be), but they provided an entertaining treat for the player who wanted to experience a crude kind of virtual reality. So successful was this early computer role-playing game that Garriott parlayed its popularity into his own game publishing house, Origin Systems, which is still producing Ultima sequels today. Not bad for a game that was originally written in Applesoft BASIC.

The Wizardry series of games, begun in 1982 by Sir-Tech Software, carried the wireframe maze concept a step farther. The first five Wizardry games took place entirely in wireframe mazes, using high resolution color graphics only for the monsters that periodically assaulted the players (see Figure 1-1). Like the Ultima games, the Wizardry series is still being produced in the 1990s and shows no signs of waning in popularity.

It would be difficult, if not impossible, to catalog all the maze games that appeared in the subsequent decade. Many of these games, such as Labyrinth and Asylum, were designed for microcomputer systems that have long ago passed on to silicon heaven. (That particular pair of games appeared originally on the Radio Shack TRS-80 Model I in the early 1980s.) Others, like Slaygon on the Atari ST and Commodore Amiga, came and went so quickly that only rabid aficionados of the genre remember them at all.

Nonetheless, dedicated microcomputer gamers have warm memories of such popular maze-style games as Electronic Art's 1986 The Bard's Tale series, which updated the Wizardry-type dungeon with detailed color graphics and, eventually, cleverly animated monsters, or SSI's "gold box" series of CRPGs, which debuted in the late 1980s with Pool of Radiance and spawned more sequels and quasi-sequels than any other game published up until that time (see Figure 1-2). For

NAME AC HP
THRENDER GRONE 1 11
BAKSHI 4 7
RHIANNON 0 7
BROTHER SEAN 2 10
DARKSTAR 7 5
PHINEAS 4 6
3,4 E 10:53
YOU ARE OUTSIDE THE CITY HALL. THE CITY CLERK WAITS INSIDE TO AWARD COMMISSIONS.

Figure 1-2 One of several dozen three-dimensional mazes from Pool of Radiance, the first of SSI's highly successful series of "gold box" CRPGs

microcomputer gamers who owned an ST, an Amiga, or an Apple IIGS during that same period, perhaps the most fondly remembered of all maze games is Dungeon Master (FTL Games), which used bitmap images and digitized sound effects to create a dungeon maze so startlingly realistic that players tend to remember it not as a game that they once played but as a place they actually visited (see Figure 1-3). Eventually, Dungeon Master was translated from the Atari, et al., to the IBM PC and its clones, which by the early 1990s had become the preeminent platforms for microcomputer games; but by that time it had spawned a series of imitations that were nearly as entertaining as the original.

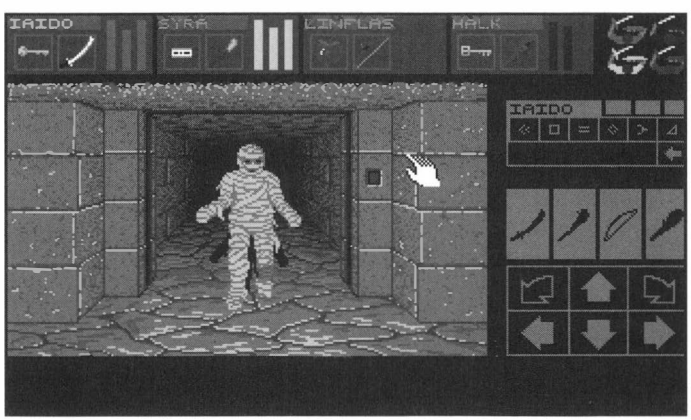


Figure 1-3 The most realistic maze game of the 1980s, Dungeon Master is regarded by many fans as the best CRPG ever published

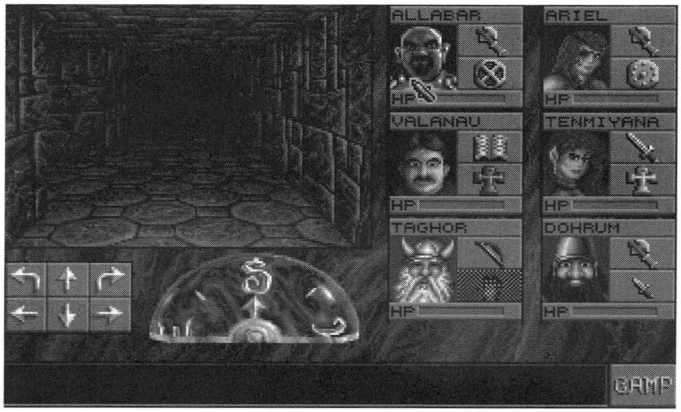


Figure 1-4 The best of the Dungeon Master clones was Eye of the Beholder, published by SSI and crafted by the programming team at Westwood Associates

The best of these Dungeon Master clones was SSI's Eye of the Beholder, from the programming team at Westwood Associates (see Figure 1-4).

Face to Face with (Virtual) Reality

For all its bone-chilling realism, Dungeon Master turned out to be merely a precursor to the super-realistic maze games of the 1990s. Dungeon Master may have featured superbly detailed maze graphics, and arguably the best mouse-driven interface ever applied to a microcomputer game, but the method used to animate the maze bore more than a passing resemblance to that used almost a decade earlier by Richard Garriott in the first Ultima. The player moved around the dungeon by pressing the arrow keys on the computer's keyboard (or clicking with the mouse on equivalent screen icons). For every press of a key, the player moved a fixed distance forward down a hallway or rotated in place by 90 degrees. All movement was in quantum leaps: It was not possible to move a shorter distance down the hallway or rotate in place by only 45 degrees. Such coarse movement controls were good enough to allow players to suspend disbelief and project themselves mentally into the dungeon, but they weren't truly realistic. In the real world, movement is smooth and fluid, not discontinuous. (Physics fans may respond that this argument breaks down when applied to particles the size of, say, electrons, but it's essentially true for large-scale objects such as human beings.)

By contrast, computer flight simulators such as Microprose's F19 Stealth Fighter and Spectrum Holobyte's Falcon have long shown game players that it is

possible to create a computer game with three-dimensional animation that is smooth and fluid, much more like movement in the real world. Armchair pilots flying the Piper Cherokee Archer in Microsoft's Flight Simulator had nearly total freedom of movement. Computer-simulated airplanes don't move ahead by sudden leaps (except on computers with sluggish CPUs). They slide through space in a continuous motion. And when they rotate, they do so by fractions of a degree. Of course, this simulated movement takes place high in the sky, with the nearest visible objects a safe distance away. Would it be possible to transfer the fluid animation of flight simulators to the interior dungeon environments of a maze game?

Yes, it would be. The first major maze game to utilize the techniques of flight simulator programming in depicting the dank interior of a dungeon was Ultima Underworld, a distant descendant of Richard Garriott's 1980 CRPG. (A much earlier Atari 800 game called Way Out, from Sirius Software, also used flight simulator techniques to create a remarkably effective maze environment, but only the most fanatic computer gamers — the author of this book, for instance — remember that this game existed.) Published in 1992 by Origin Systems, Ultima Underworld allowed the player to crawl through the slimy environs of a vast underground complex, all of it depicted with flight simulator-style animated graphics. While screen shots from the game, such as the one in Figure 1-5, may not look as realistic as those from Dungeon Master or Eye of the Beholder, the smoothness of the animation more than compensated for any shortcomings. The Underworld dungeon may not have looked real, but it *felt* real, with the kind of pit-of-the-stomach believability that computer games very rarely achieve. Ultima Underworld brought the maze game into the age of virtual reality.

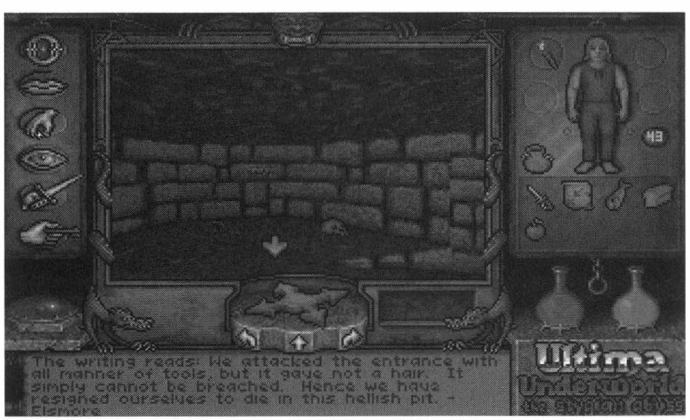

The writing reads: We attacked the entrance with all manner of tools, but it gave not a hair. It simply cannot be breached. Hence we have resigned ourselves to die in this hellish pit. - Elsmore
Ultima
Underworld

Figure 1-5 Ultima Underworld brought the maze game into the age of virtual reality

A Maze in Wolf's Clothing

Ultima Underworld may be the most realistic and detailed maze game to come down the pike so far, but only a few months after it was released, another maze game appeared that made even seasoned Underworld players shake their heads in astonishment. Called Wolfenstein 3D and marketed as shareware by Apogee Games, it featured a maze environment that was in some ways less believable, and certainly far less detailed, than the one in the Origin game (see Figure 1-6). But what it lacked in realism, it more than made up for in sheer velocity. Wolf 3D, as it is popularly known, was by all odds the *fastest* maze game ever published. Like Underworld, it allowed the player to move smoothly and fluidly through a maze of rooms and passages, but at a far more hectic pace. Wolf 3D was pure action. Tossing aside most of the role-playing features of earlier maze games, Wolf 3D was a compulsively playable arcade game.

It was also the first of a new genre of maze game. The programmers at Id Software, who had developed Wolf 3D for Apogee, marketed the animation engine from that game to other programmers who wanted to produce their own Wolf 3D-style games. One of the first of these, Blake Stone: Aliens of Gold (written by Jam Software), was released by Apogee in late 1993 (see Figure 1-7). In the meantime, the programmers at Id proceeded to develop a more advanced version of the Wolf 3D animation engine, with more features than the original. This engine became the basis for a game called Doom (see Figure 1-8), published in December 1993, which took place in a far more realistic maze environment than the one in Wolf 3D. Doom introduced lightsourced 3D graphics and heightmapping to the Wolf 3D-style maze game. Translated from computerese,

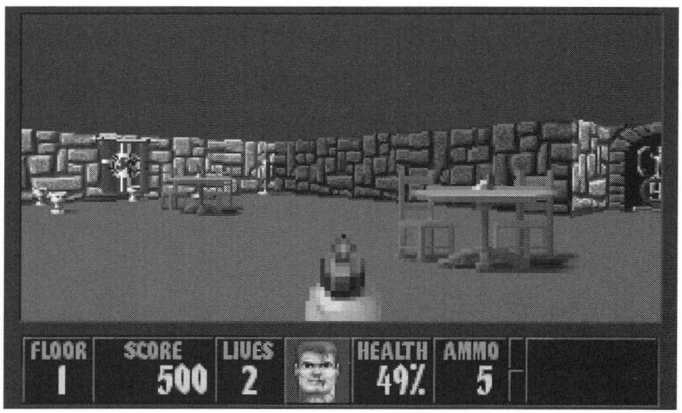


Figure 1-6 The wildly popular Wolfenstein 3D introduced a new type of maze game that combined elements of CRPG and arcade game with smooth three-dimensional animation

Figure 1-7 Blake Stone: Aliens of Gold uses the animation engine developed for Wolfenstein 3D

that means that the maze in Doom is illuminated by changing and moving light sources and that the floor rises and falls. Stairs lead from one dungeon level to the next. Walls fade in and out of view as they move closer to and farther away from the player's lantern. Lights blink on and off. (Much of this was also true of Ultima Underworld, but Wolf 3D used a different — and potentially faster — style of animation than Underworld, as we'll see later in this book.)

Inspired by the success of Wolf 3D, more "smooth-scrolling arcade games" (as they are sometimes called) appeared in 1993, including Shadowcaster from Origin Systems, which also used the Id animation engine. Bethesda Softworks

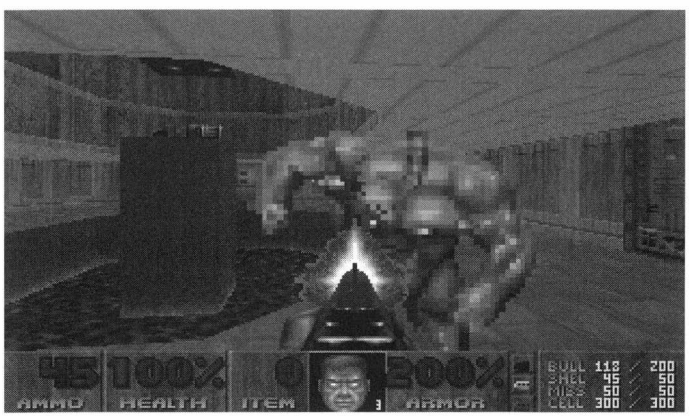


Figure 1-8 Although written by the same programmers who produced Wolfenstein 3D, Doom features even more sophisticated and innovative maze graphics techniques

Figure 1-9 Terminator: Rampage is a maze game based on the popular series of films

published a maze game called Terminator: Rampage (based on the popular film series); in 1993 (see Figure 1-9) and The Elder Scrolls: Arena in early 1994 (Figure 1-10). Westwood Associates, now with their own label separate from SSI, released a Dungeon Master-style CRPG called Lands of Lore in the summer of 1993.

As the mid-1990s neared, it was clear that the maze game was the hottest thing going in the world of computer games. This left a lot of computer game programmers scratching their heads and asking the inevitable question: Just how do you write one of these games, anyway?

Figure 1-10 The Elder Scrolls: Arena applies Wolf 3D-style animation techniques to the CRPG game

Types of Mazes

The short answer to that question is that there are several different ways to write a maze game. In fact, it could be argued that there are as many different ways of writing a maze game as there are programmers writing them.

However, we can break the vast majority of maze games down into four basic types, each of which is programmed in a different manner. In this book, we'll refer to those types as wireframe mazes, bitmapped mazes, polygon mazes, and ray-cast mazes. (It's also possible to divide maze games into genres such as CRPGs and arcade games. In this book, however, we'll be talking mostly about the technical details of implementing such games, not the higher-level details of game design. This should not be taken to mean that game design is not important, just that it deserves a great deal more space than I would be able to give it in this book.)

Each of the maze games mentioned earlier in this chapter falls into one of these four categories:

- Wireframe mazes: The early Ultima and Wizardry games
- Bitmapped mazes: The Bard's Tale series, Dungeon Master, the SSI "gold box" games, Eye of the Beholder, the later Wizardry games, Lands of Lore
- Polygon mazes: Ultima Underworld 1 and 2
- Ray-cast mazes: Wolfenstein 3D, Doom, Blake Stone: Aliens of Gold

Although some of these types of maze game may look superficially similar to the player, from the programmer's point of view each requires a very different approach.

Wireframe Mazes

Wireframe mazes are the simplest type of maze to program and require the least CPU power and memory resources to animate, which is why this approach was used in most of the early microcomputer maze games. All the programmer needs in order to create this type of maze is a fast line-drawing routine and the patience to figure out where to put the lines on the display. The programming techniques required for creating a wireframe maze are simple and straightforward, and no advanced mathematics are required. We'll discuss wireframe maze techniques in some detail in the next chapter.

Bitmapped Mazes

Bitmapped mazes aren't much more difficult to program than wireframe mazes. In fact, it could be argued that they're a little easier, since no line-drawing routine is required. Bitmapped maze graphics are pieced together from predrawn images of walls, floors, ceilings, and bits of set decoration. The only catch is that a competent artist is required to draw these images (though in some instances a ray-tracing program will suffice). Once drawn, getting the images on the video display is a piece of cake, relatively speaking. We'll discuss bitmapped mazes in more detail in chapter 4.

Polygon Mazes

Polygon mazes are the most difficult type to program. The techniques used to animate such mazes are essentially the same as those used to animate microcomputer flight simulators, and they require at least a passing knowledge of trigonometry and geometry on the part of the programmer. The results, however, can be worth it. Polygon mazes are potentially more realistic than the mazes created using other methods, but they require the programmer to overcome a series of difficult technical problems, including hidden surface removal, polygon clipping, three-dimensional database design, and texture mapping. Ironically, the images in a polygon maze program are often less realistic than those in a bitmapped maze game, but the greater realism of continous, flight simulator-style movement more than makes up for this lack of realism. We'll talk about polygon mazes at some length in chapters 5 and 6.

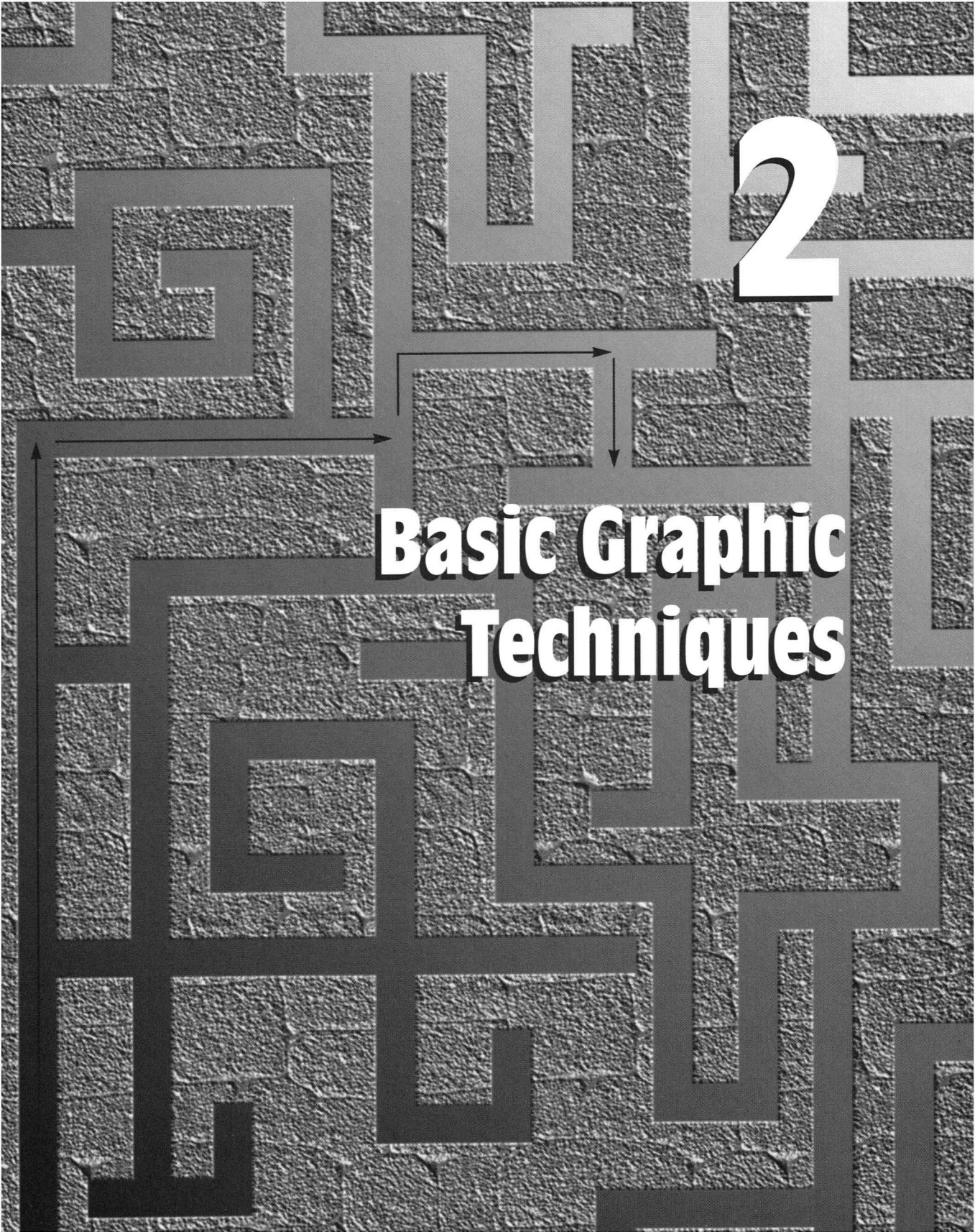
Ray-Cast Mazes

Ray-cast maze techniques are surprisingly easy to master and can produce effects almost as vivid and realistic as those found in polygon maze games. Ray-casting techniques use simple principles of optics to project a detailed image of the floor, walls, and ceiling of a maze onto the video display of a computer. These techniques resemble an extremely simplified version of the ray-tracing techniques that are used to produce highly realistic animated images for movies and TV. (We'll discuss ray tracing at greater length in chapter 8.) The only drawback of ray casting as a method of drawing mazes is that the ways in which the player is allowed to move through the maze must be limited somewhat and the types of objects that the player can encounter must be somewhat reduced relative to a polygon maze game. However, in a cleverly designed maze game, most players will never notice these limitations. Player movement in a ray-cast maze game can

be every bit as fluid and realistic as in a polygon maze game, and in many cases the animation is even smoother. This combination of programmer ease and high-velocity animation makes ray casting the ideal method for creating arcade-style maze games on current generation microcomputers. For that reason, we'll spend the last five chapters of this book discussing ray-casting techniques and developing a ray-casting animation engine that can be used to produce both arcade-style maze games and CRPGs. We'll even develop a game that uses this engine.

Animation Programming

In the chapters that follow, we'll recapitulate the evolution of the maze game, exploring the whole gamut of techniques used to render more and more realistic mazes on the microcomputer video display. Before we can talk about maze game programming, however, we'll need to say a few words about computer graphics and animation in general. In the next chapter, we'll discuss the basics of producing images on the video display of an IBM-compatible microcomputer — then we'll use those techniques to create the rudiments of a wireframe maze game, similar in visual spirit to the early Wizardry games.


2
Basic Graphic
Techniques

rogramming the video display of a microcomputer isn't difficult. Most computer programming languages come with either a built-in set of commands or a standard library of functions for putting information on the display. The problem is, these commands and functions aren't necessarily up to the demands that a game programmer is likely to place on them. There are libraries of functions available that have been optimized for game programming needs, and many game programmers find these libraries indispensable, but they will cost you money above and beyond the small fortune that you've already laid out to buy your compiler or interpreter.

In this book we'll develop our own set of graphics functions in Borland C++ and assembly language for outputting graphics to the video display of an IBM-compatible microcomputer. The Borland compiler comes with a graphics library called the BGI (for Borland Graphics Interface), but the functions included in it aren't fast enough or versatile enough to handle the tasks that we're going to throw at them. If you have a favorite library of graphics functions that you'd prefer to use, feel free to substitute them where necessary in your own code. However, writing graphics code from scratch isn't a lot more difficult than learning to use a prewritten library of functions — and the graphics code that you write yourself is guaranteed to do exactly what you want it to do.

After you've finished debugging it, anyway.

Video Memory: A Brief Tour

The major concept that you need to understand in order to program graphics on an IBM-compatible microcomputer (or just about any other type of microcomputer, for that matter) is the concept of video memory. Inside your microcomputer, usually in one of the slots attached to the motherboard, is a set of electronic circuits that store an encoded version of the image currently on the video display. Change the contents of these circuits and you change the image. Programming graphics is as simple — and as complicated — as that.

Up until now, you may not have delved into the esoteric rules of programming a computer on the level of bits and bytes. But it's time to lose your innocence. Programming your computer's memory directly — especially the video memory — may sound intimidating, but it's simple once you learn the rules.

Offsets and Addresses

Every type of information that's stored inside your computer — programs and databases, graphics and musical scores — is stored in memory. Yet memory is nothing more than a series of electronic circuits, each of which can store an electrical representation of a binary number — that is, a number made up of the digits 0 and 1. We don't have to worry precisely how these numbers are represented electrically or even how the binary numbering system works, because most programming languages will do the work of translating other forms of information into the internal form required by the computer and not bother us with the details. However, it helps to understand the essential layout of the computer's memory if you're going to put images on the screen.

Each *memory circuit* can hold a number in the range 0 to 255. (I'm representing these numbers using the decimal numbering system, because that's the only system that all readers of this book are guaranteed to be familiar with, but the memory circuits store them in binary.) There are commands in the C++ language, as in most other programming languages, that will place a number in a memory circuit or retrieve a number that's already stored in a circuit.

Because it's important that data be retrieved from the same memory locations in which it has been stored, each location has an identifying number, called an address, that can be used to distinguish it from other locations. It's traditional to write these addresses using the hexadecimal numbering system. Hexadecimal has strong mathematical ties to binary, which makes it easy to translate from one system to the other; however, it takes fewer digits to represent a number in hexadecimal than in binary, making it by far the less unwieldy of the two systems. Hexadecimal numbers use 16 different digits, the numerals 0 through 9

plus the letters A through F. The number 65535 in hexadecimal, for instance, is FFFF. In C++ or C, this would be represented as 0xffff, with the "0x" identifying the number as hexadecimal.

The addresses used to identify memory locations in IBM-compatible microcomputers are made up of two parts: a segment and an offset. The reasons for this are complex and have to do with the nature of the IBM hardware; we're not going to go into what they are. Each of these numbers, the segment and the offset, can fall anywhere in the range 0000 to ffff hexadecimal — that's 0 to 65535 in decimal. The address is written by separating the segment and offset with a colon, like this: 1234:5678. Such a segment-offset pair can also be translated into a *linear address* by multiplying the segment by 16 and adding it to the offset, though you'll rarely, if ever, need to do this. It's important to bear in mind, though, that two segment-offset pairs that translate into the same linear address represent the same location in the computer's memory, even though the segment and offset numbers may be different. Thus 0000:0010 and 0001:0000 represent the same memory location, because they both translate to the linear address 00010 hexadecimal (or 16, in decimal).

Manipulating Memory with Pointers

A C++ programmer can use several different methods to put a numeric value into a memory location. Every time the value of a variable is changed, for instance, one or more numbers are placed in memory locations that have been assigned to that variable by the C++ compiler. To interact with video memory, however, the programmer needs to be able to change the value of specific memory locations rather than locations chosen arbitrarily by the compiler.

The most direct way to change the value of a specific memory location is to use a C++ pointer. A pointer is a variable that represents the address of a location in the computer's memory. In most versions of C++ written for IBM-compatible microcomputers, pointers come in two varieties: near pointers and far pointers. The primary difference between the two is that the programmer can control precisely which memory location a far pointer represents, but can only control the offset of the memory location represented by a near pointer. The segment portion of the address pointed to by a near pointer is determined by the operating system at the time the program is loaded. Since we need to control both the segment and offset portions of the address in order to interact properly with video memory, we'll be using far pointers exclusively in this book.

When declaring a pointer, the programmer must tell the compiler what sort of values it will be pointing to in the computer's memory — *int, char, float,* and so on. In programming video memory, we'll need to point to byte-sized values, so

we'll use pointers to values of type *char*. A far pointer to values of type *char* is declared like this:

```
char far  *charpointer;
```

This declares a far pointer called *charpointer,* which points to values of type *char*. To set *charpointer* equal to a specific memory address, we use the MK_FP macro, which is defined in the Borland header file DOS.H. In order to use it, you'll need to include DOS.H at the top of your program file.

The MK_FP macro — the name stands for "make far pointer" — is used like this:

```
charpointer = (char far *) MK_FP(0x7456, 0x0890);
```

This assigns the address 7456:0890 to the pointer *charpointer*. The typecast in parentheses in front of the MK_FP macro converts the pointer, which is a far pointer to type *void,* into a far pointer to type *char*.

After all this work, we're finally ready to place a value in the memory location pointed to by the pointer. To do so we must *dereference* the pointer, which is done using the * or *dereferencing operator,* like this:

```
*charpointer = 179;
```

This assigns a value of 179 to the memory location 7456:0890, assuming that the value of *charpointer* has not been changed since the previous assignment statement. To retrieve this value, we can put the dereferenced pointer on the other side of the assignment operator:

```
char thisvar = *charpointer;
```

This assigns the value stored at the address pointed to by *charpointer* to the *char* variable *thisvar.* (You'll notice that we've declared the *char* variable *thisvar* in the assignment statement itself. This is allowed in C++ at the time of a variable's first use, though it's important to note that the variable will only be valid in the block where it is declared. For instance, if this statement is placed inside a loop, the variable *thisvar* will no longer be valid when the loop terminates. In technical lingo, it will "go out of scope.") If nothing occurs to change the value at that location between these two assignment statements, *thisvar* will be set equal to a value of 179.

The dereferenced pointer **charpointer* can be used anywhere within a program that a *char* variable could be used. To print the value stored at the location pointed to by *charpointer* as a signed decimal integer, for instance, you could use the *printf* statement:

```
printf("%i\n",*charpointer);
```

Mode 13h

So what memory locations do you need to change in order to program video memory? And what values do you need to change them to?

That depends on several things. Obviously, it depends on what sort of image you want to place on the display. But it also depends on what video mode the computer is currently in.

The programs in this book are designed to run on the IBM VGA video adapter, variations on which can be found in the vast majority of current PCs. The VGA adapter can be operated in 24 different "official" modes and quite a few more unofficial ones. Some of these are text modes, which means that the numeric values represent text characters. The remaining modes are graphics modes. When the VGA adapter is placed in a graphics mode, the values placed in video memory represent the colors of the tiny dots on the video display, known as pixels, which make up all graphic images.

In this book we'll use the VGA adapter in the "official" graphics mode known as mode 13h. (The "h" after the 13 indicates that the number is in hexadecimal. In decimal this mode is known as mode 19, but the hexadecimal identifier is more commonly used than the decimal one.) In this mode all graphics are constructed out of a matrix of 320 pixels horizontally and 200 pixels vertically. Each of these pixels can be in any of 256 different colors (which can, in turn, be chosen from a list of more than 256,000 different colors).

Mode 13h is the most common mode for programming animated video games on the PC. There are a number of reasons for this, the main one being that it is the only standard VGA mode that supports 256 simultaneous colors on the display. Some VGA adapters offer additional 256-color modes — quite a few even offer modes in which 16 million different colors can be simultaneously present on the display — but none of these modes is standardized. They are programmed differently on different brands of adapters, and on many adapters they are simply not available. Furthermore, most of these modes feature higher resolutions — numbers of pixels — than mode 13h. Although this improves the appearance of the graphics, it also tends to slow down the speed of the animation, since a much larger amount of data must be moved around in the computer's memory in order to draw and animate an image. Games that use the higher-resolution 256-color modes are already appearing on the market, but for the moment they are still outnumbered by lower-resolution games. A year or two from now, one of these modes may be the standard mode for video game development, but at present it is mode 13h that has that honor. Fortunately, most of the techniques that we'll discuss in this book for animating images on the mode 13h display can be used in higher-resolution modes with only minor changes to the program code.

Setting the Video Mode

To make our job easier, the code for turning mode 13h (or any other "official" VGA video mode) on and off is built into the video card itself. We can use this code in a program by going through the built-in ROM BIOS, the set of program routines that are installed in the computer at the factory. However, many of the ROM BIOS routines — including the ones for setting the video mode — are best accessed through assembly language modules rather than through high-level programming languages such as C++, so in this book we're going to create a small library of assembly language routines for communicating with the ROM BIOS (as well as with video memory itself) and put them in a file on the accompanying disk called SCREEN.ASM. We'll then link this file directly with our C++ programs, so that we can call these routines from our C++ code as though they were C++ functions. This file will be automatically assembled by Turbo Assembler (TASM), a utility program that comes as part of the full Borland C++ 3.1 package, when its name is included in a Borland PRJ file. (For those readers who lack the full Borland C++ package with TASM, the already assembled version of this code is also included on the accompanying disk under the file name SCREEN.OBJ. Substitute this name for SCREEN.ASM in the project files or make files for all programs in this book that reference it. TASM may be purchased separately for readers using 4.0.)

There isn't space in this book for a full-scale primer on assembly language, though we'll return to the subject at somewhat greater length in chapter 12, when we discuss program optimization. (A brief overview of assembly language programming is provided in Appendix B.) And you don't really need a knowledge of assembly language to use these functions (though it's helpful to know an assembler if you want to revise these functions or write some of your own). So that you won't be totally in the dark about how these functions work, I'll explain how some of the assembly language instructions and conventions work as they arise, but it's not necessary that you grasp every little detail of this code.

When we write an assembly language routine that can be called as a C++ function, we must begin it with a PROC statement:

```
_setmode  PROC
```

This tells the assembler (the program that translates this assembly language code into a form that can be linked with a C++ program) that we're about to create an assembly language procedure called *_setmode*. (When we write an assembly language procedure that is to be called by name from a C/C++ program, the name of that procedure must begin with an underscore. However, when we call it from C++, we must drop the underscore and refer to it by the name *setmode()*, without the underline.) We can then use the assembly language *ARG* directive to

tell the assembler that we'll be passing a single parameter to this procedure when we call it:

```
ARG    mode:WORD
```

This is equivalent to declaring a function in C++ with the function header *setmode(int mode)*. Now we can reference the label *mode* in the text of the procedure, and the assembler will translate that reference into code referring to the value that we pass from C++. However, for this to work correctly, we must first include a piece of boilerplate code to take care of the technical details:

```
push   bp
mov    bp,sp
```

Don't worry about what this does. Just remember that, without it, the *ARG* directive won't work properly.

We then must move the value of the *mode* parameter into AX, which is a special memory location within the computer's CPU known as a *register:*

```
mov    ax,[mode]
```

Translated into something resembling English, this instruction means "move the value in the memory location that we have labeled *mode* and place it in the special memory location known as AX." Now that the video mode is in location AX, we must place a 0 in the location known as AH:

```
mov    ah,0
```

Memory location AH is actually part of memory location AX, though you don't really need to know that in order to understand what this code does. Basically, we are telling the ROM BIOS that it should execute video function 0 (which is the function that sets the VGA mode) to put the video display into the mode stored in AX, which will be equal to the parameter we passed from C++. Then we execute this instruction:

```
int    10h
```

This calls the ROM BIOS itself to do the actual work of changing the video mode. All that's left to do is clean up the mess we made with the instructions that set up the computer's memory for the *ARG* directive:

```
pop    bp
```

Finally, we must execute a RET instruction to tell the CPU that it's time to return control from this function back to the program that called it:

```
ret
```

And we must tell the assembler that the assembly language procedure is finished, with the ENDP directive:

```
_setmode  ENDP
```

The complete SETMODE procedure appears in Listing 2-1.

Listing 2-1 The SETMODE procedure

```
; setmode(int mode)
;   Set VGA adapter to BIOS mode MODE

_setmode  PROC
  ARG     mode:WORD
  push    bp          ; Save BP
  mov     bp,sp       ; Set up stack pointer
  mov     ax,mode     ; AX = video mode
  mov     ah,0        ; AH = function number
  int     10h         ; Call video BIOS
  pop     bp          ; Restore BP
  ret
_setmode  ENDP
```

The semicolons in several of the program lines indicate that what follows is a comment to be ignored by the assembler. Because assembly language code tends to be rather obscure, it's a good idea to embed such comments wherever possible, to clarify the programmer's intent.

To put the VGA adapter into mode 13h, you can call this function from C++ like this:

```
setmode(0x13);
```

Finding Video Memory

Once you've put the VGA adapter into mode 13h, finding video memory is a piece of cake. It always begins at address A000:0000 (which is equivalent to a linear address of A0000 hexadecimal) and continues for a total of 64,000 (or F800 hexadecimal) addresses thereafter. The last address in mode 13h video memory is A000:F7FF. (The remaining 1,535 bytes of the 64-kilobyte segment video memory are unused, since a segment is 65,536 bytes long and there are only 64,000 pixels on the mode 13h display.)

Changing the value stored in any of the addresses in this range will change the color of the pixel in the corresponding position on the video display. How can you tell which pixel corresponds to which memory address? In mode 13h the pixels on the display are arranged in 200 horizontal rows, each containing 320

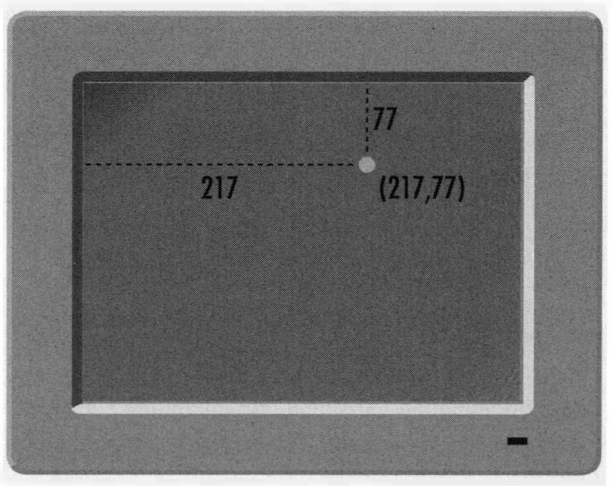


Figure 2-1 Each address in video memory corresponds to a pixel on the video display

pixels. The first location in video memory — that is, the location designated by the address A000:0000 — corresponds to the first pixel in the first row of pixels at the top of the display, which is to say the pixel in the upper-left corner of the screen. The second location — the one designated by address A000:0001 — corresponds to the pixel immediately to the right of that pixel. And so on for the first row of the display, with the location designated by address A000:013F corresponding to the pixel in the upper-right corner of the display. The next 320 memory locations correspond to the 320 pixels in the row immediately below this one, so that address A000:0140 corresponds to the pixel directly below the one in the upper-left corner. Each succeeding 320 addresses in video memory corresponds to another row of pixels on the display, moving from the top row sequentially to the bottom one. The last address in video memory, A000:F7FF, corresponds to the pixel in the lower-right corner of the display (see Figure 2-1).

With this information we can write a short program to paint the mode 13h video display entirely white. Such a program appears in Listing 2-2.

Listing 2-2 The WHITEOUT.CPP program

```
//
// WHITEOUT Version 1.0
//
// Fills mode 13h video memory with a constant value of 0x0f,
// filling the display with white pixels
//
// Written by Christopher Lampton
//  for Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  "screen.h"

void main(void)
{

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Create pointer to video memory:
  char far *screen=(char *)MK_FP(0xa000,0);

  // Set mode 13h:
  setmode(0x13);

  // Fill video memory with a constant value:
  for (unsigned int i=0; i<64000; i++) screen[i]=0x0f;

  // Wait for user to press a key:
  while (!kbhit());

  // Reset original video mode:
  setmode(oldmode);
}
```

At the beginning of the program listing, after the initial comments, is a list of the header files to be included. The SCREEN.H header file is on the accompanying disk and must be included in all programs that use the functions in the SCREEN.ASM file. The others are standard Borland headers. The program itself consists of a single *main()* function. The first line of this function reads the value of the previously set video mode, which is normally stored at memory address 0040:0049, into the *int* variable *oldmode*, so that the mode can be restored later. (If the old mode was never restored, the user would exit the program to find that the display was still in mode 13h.)

The MK_FP macro is then used to point the pointer variable *screen* to the start of video memory. The *setmode()* function, which was introduced a few pages ago, is called to set the screen to mode 13h. At that point, all that's needed to paint the display white is a simple *for()* loop, which iterates the unsigned integer variable *i* through all 64,000 offsets in the video memory segment. We can access these locations by treating the pointer variable *screen* as an array of *char* beginning at address A000:0000. The array offset *i* — which should not be confused with the offset portion of a memory address, though its function is similar — is added to the address of the start of the array, so the successive values of *i* cause the assignment statement we've placed in the loop to place the constant value 15 throughout video memory. The last few lines wait for the user to press a key (using the *kbhit()* function from the CONIO.H file) and restore the old video mode (using the same *setmode()* function that we used to set mode 13h).

Run the program by typing WHITEOUT at the DOS prompt and watch what happens: The screen fills rapidly with white pixels. If you have a fast machine, it will fill so quickly with this single color that the change will appear nearly instantaneous.

The VGA Palette

Why does the screen fill with the color white and not, say, mauve or chartreuse? The key is in the program statement

```
for (unsigned int i=0; i<64000; i++) screen[i]=0x0f;
```

This is the statement that assigns a value of 0x0f (or decimal 15) to each memory location within mode 13h video memory. Try changing this number to some other number in the range 0 to 255 (hexadecimal 00 to FF) and recompiling the program. If you change the number to 0x0e (decimal 14), the screen will fill with the color yellow instead of white. If you change the number to 0x1b (decimal 27), the screen will fill with the color gray. And so forth.

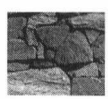
Each of these numbers, when placed in a mode 13h video memory location, causes a different color of pixel to appear on the display. How do we know what numbers correspond to what colors of pixel? The VGA adapter keeps a list of 256 colors, each of which corresponds to one of the 256 possible values that can be stored in each video memory location. This list of colors is called the palette. The first color in the palette list corresponds to a value of 0 stored in video memory, the second color in the list corresponds to a value of 1, and so forth — up to the 256th color in the list, which corresponds to a value of 255. As it happens, the sixteenth color in the list is white, which is why placing a value of 15 (which corresponds to the sixteenth color in the palette) in video memory causes a white

pixel to appear on the display. The fifteenth color in the list is yellow, which is why placing a value of 14 in video memory causes a yellow pixel to appear on the display. The twenty-eighth color in the list is gray. And so forth.

As you might guess, there's a built-in function in the VGA BIOS that allows us to change the colors in the palette list. Listing 2-3 contains a short assembly language procedure (which can be found on the disk in the file SCREEN.ASM) that can be called from C++ to set the palette. The syntax for calling this function is

```
setpalette(color_regs);
```

where *color_regs* is a far pointer to an array of *char*. This array must have 768 elements in it. The first three elements in this array contain a color descriptor for the first color in the palette, the second three elements contain a color descriptor for the second color in the palette, and so on. Each of these color descriptors consists of three values in the range 0 to 63, where the first value represents the blue intensity of the color, the second value the green intensity of the color, and the third the red intensity of the color. A value of 0 indicates minimum intensity in that component of the color, and a value of 63 represents a maximum intensity in that component; all other values represent an intensity between these extremes. Thus a color descriptor for a pure, bright shade of red would consist of the values 0,0,63. A moderate shade of gray would consist of the values 32,32,32. The color white would consist of the values 63,63,63. And so on.



Listing 2-3 The SETPALETTE procedure

```
; setpalette(char far *color_regs,intfirstreg,int numregs)
;   Set VGA color registers, beginning with FIRSTREG and
;      continuing for NUMREGS to the color values in COLOR_REGS

_setpalette    PROC
    ARG   regs:DWORD
    push  bp              ; Save BP
    mov   bp,sp           ; Set up stack pointer
    les   dx,regs         ; Point ES:SX at palette registers
    mov   ah,10h          ; Specify BIOS function 10h
    mov   al,12h          ; ...subfunction 12h
    mov   bx,0            ; Start with first register
    mov   cx,100h         ; Set all 256 (100h) registers
    int   10h             ; Call video BIOS
    pop   bp              ; Restore BP
    ret
_setpalette    ENDP
```

When you turn the computer on, the VGA BIOS automatically sets up a default palette. Listing 2-4 contains a short program that will display the default mode 13h palette as a 16 by 16 matrix of small squares, with the first row of 16 squares representing the first 16 colors of the palette, the second row of 16 squares representing the next 16 colors of the palette, and so on. This program, which can be found on the disk as PALETTE.CPP (and executed by typing PALETTE at the DOS prompt), is nearly identical to the earlier WHITEOUT.CPP program. The only difference is in the *for()* loop, which is now a nested sequence of *for()* loops. The outermost loop runs through all 256 palette-colored squares. The two inner loops cycle through the 12 vertical and 20 horizontal pixels in each square. The assignment statement that sets the pixels uses the division (/) and modulus (%) operators to calculate the video memory offset corresponding to the upper-left corner of the square, then adds in the *x* and *y* positions of the pixels within the square to get the exact memory address to change. We'll talk more about this technique of identifying video memory locations in a moment, so you might want to come back and look at this program again after you've finished reading this chapter. The output from the PALETTE program is shown (as a gray-scale image) in Figure 2-2.

Listing 2-4 The PALETTE.CPP program

```
//
// PALETTE Version 1.0
//
// Draws the mode 13h default palette as a matrix of
//  squares on the video display
//
// Written by Christopher Lampton
// for Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  "screen.h"

void main(void)
{

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Create pointer to video memory:
  char far *screen=(char *)MK_FP(0xa000,0);
```

continued on next page

continued from previous page

```
  // Set mode 13h:
  setmode(0x13);

  // Draw a 16x16 matrix of squares in the default colors
  for(int square=0; square<256; square++)
    for(int y=0; y<12; y++)
      for(int x=0; x<20; x++)
        screen[square/16*3840+square%16*20+y*320+x]=square;

  // Wait for user to press a key:
  while (!kbhit());

  // Reset original video mode:
  setmode(oldmode);
}
```

A Screen-Clearing Function

We need to add one more function to our SCREEN.ASM file before we can start doing some serious graphics coding. It will be a function for clearing the video display, to eliminate any trash that might have been left there by earlier programs (or to clear one screen off the display to make room for another). It would be easy enough to adapt the earlier WHITEOUT program for this purpose, changing the value of 15 that WHITEOUT stores in video memory to a value of 0 (which represents the color black in the default palette). But to guarantee that we're getting maximum speed and efficiency for our programming money, we'll write this function in assembly language instead.

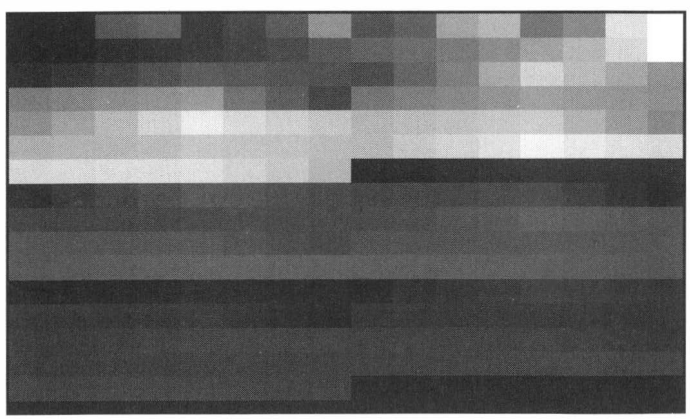


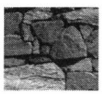
Figure 2-2 The output from the PALETTE.CPP program, showing all 256 colors in the default palette (here represented in gray-scale)

The completed function appears in Listing 2-5. It's called CLS, short for CLear Screen. It can be called from C++ like this:

```
cls(screen_address)
```

The parameter *screen_address* is the address of the first location in video memory. You might wonder why we don't simply hardcode this address into the function itself, since mode 13h video memory is always located in the same place. Later, however, we'll want to draw our screen images in locations other than video memory, then move them all at once to video memory. Passing a screen address parameter to *cls()* function will allow us to do this without writing a whole new battery of functions.

The key instruction in the program listing is REP STOSD. This is one of the so-called string instructions offered by the 80x86 series of microprocessors (the series that all IBM-compatible microcomputers are based on). This instruction fills the entire video display, by itself, with the value stored in the AX memory register. Before it executes, the program places a 0 in this register, so that the video display is cleared to black. Assembly language programmers will recognize that this particular string instruction is one that will only work on 80386 or better microprocessors, because it moves data 32 bits (that is, 4 bytes) at a time. We're using this instruction because state-of-the-art VGA boards are connected to the computer's CPU via 32-bit interfaces and the REP STOSD instruction will execute twice as fast as the 16-bit REP STOSW instruction that is more typically used to perform this task. (On older graphics boards, with 8- or 16-bit interfaces, it probably won't provide any noticeable improvement in speed over the 16-bit instruction.)



Listing 2-5 The CLS procedure

```
; cls(char far *screen_adr)
;   Clear video memory or offscreen buffer at
;   SCREEN_ADR to zeroes

_cls  PROC
ARG screen:DWORD
  push    bp            ; Save BP
  mov     bp,sp         ; Set up stack pointer
  push    di            ; Save DI register
  les     di,screen     ; Point ES:DI at screen
  mov     cx,SCREEN_WIDTH/4 * SCREEN_HEIGHT ; Count pixels
  mov     ax,0          ; Store zero values...
  rep     stosd         ; ...in all of video memory
  pop     di            ; Restore DI
```

continued on next page

continued from previous page

```
  pop     bp              ; Restore BP
  ret
_cls   ENDP
```

Building a Wireframe Maze

As promised in the last chapter, we are now going to develop a program that builds a wireframe representation of a maze on the computer's video display. The trickiest part of drawing a wireframe representation of a maze is drawing the wires themselves. What we need is a function that will connect any two pixels on the video display with a line. Although the Borland Graphics Interface has a built-in line drawing function, it doesn't work in mode 13h (and isn't necessarily fast enough for our purposes anyway). So we'll implement our own line-drawing function. Even if you plan to use the line-drawing function in your favorite graphics library instead, you shouldn't skip over the description of how our line-drawing function performs its magic, since we'll be using some of the same techniques in later chapters to scale bitmaps.

The Cartesian Coordinate System

Video memory, as we have seen, is a series of memory locations corresponding to the pixels on the video display in row-and-column order, with the top row of 320 pixels on the display corresponding to the first 320 locations in video memory, the second topmost row of 320 pixels on the display corresponding to the second 320 locations in video memory, and so forth. However, this isn't always the best way to visualize either video memory or the video display. Throughout this book we'll lean heavily on a more abstract but useful system for identifying the positions of pixels on the screen — the x and y coordinates of a Cartesian coordinate system.

The Cartesian coordinate system, invented by the French philosopher René Descartes in the seventeenth century, is commonly used in algebra and geometry for the graphing of equations. If you've ever taken a course in high school level mathematics, you've probably run across it. The Cartesian coordinate system identifies points on a plane by their positions relative to a pair of lines, one horizontal and one vertical, on which individual points correspond to numbers. These lines are called the horizontal and vertical axes (plural of axis) of the coordinate system (see Figure 2-3).

Each point on the Cartesian plane — the flat area defined by the two axes of the coordinate system — can be identified by a pair of numbers referred to as a *coordinate pair*. To learn what coordinate pair corresponds to a given point, you

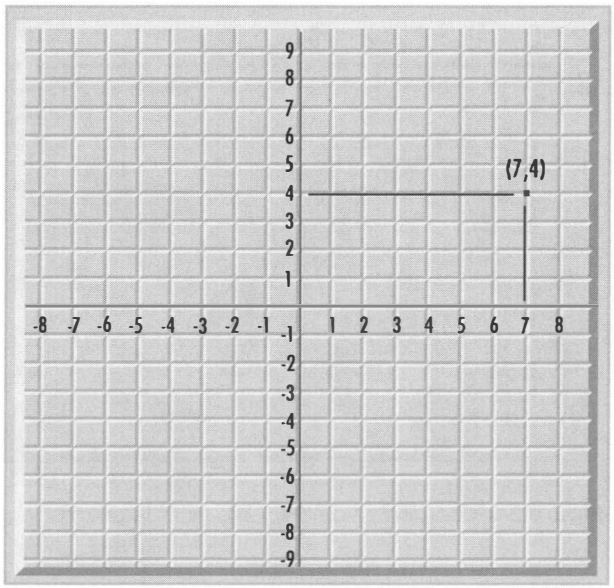


Figure 2-3 A Cartesian system defined by horizontal and vertical axes

must draw a vertical line from that point to the nearest point on the horizontal axis, and a horizontal line from that point to the nearest point on the vertical axis. The points at which these lines cross the horizontal and vertical axes, respectively, are called the x and y coordinates of the point. The point in Figure 2-3, for instance, has x and y coordinates of 7 and 4.

The computer screen is more or less flat, so we can imagine it as a kind of Cartesian plane, with the vertical axis running along the left edge of the display and the horizontal axis running along the top. In a traditional Cartesian coordinate system, the coordinates run from left to right on the horizontal axis (which is usually referred to as the x axis), and from bottom to top on the vertical axis (which is usually referred to as the y axis). But when describing a computer display in Cartesian terms, the vertical axis is usually flipped upside down so that the numbers run from top to bottom, to correspond to the way in which video memory addresses grow larger when moving down the display (see Figure 2-4). This makes the translation from Cartesian coordinates to video addresses a little simpler.

Plotting Pixels

If the video display is a Cartesian coordinate system, then every pixel on it can be described by a pair of x,y coordinates, relative to the horizontal and vertical axes

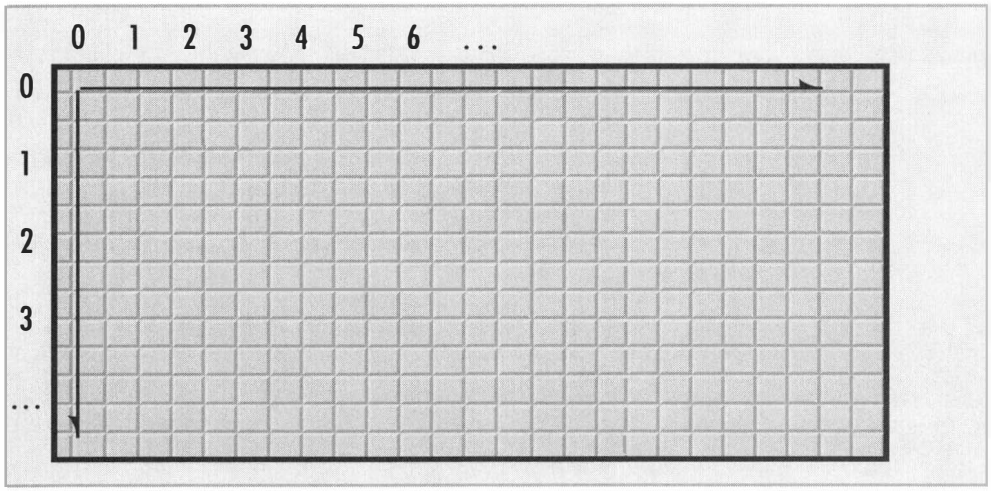


Figure 2-4 Coordinates on the y axis grow larger moving down the display

that we described in the last paragraph. (The two axes don't necessarily have to be in the positions that I ascribed to them. Although these particular coordinate axes are the ones traditionally used to describe the video display, we can move the axes pretty much anywhere on or off the display, as you'll see later in this book.)

In the Cartesian coordinate system depicted in Figure 2-5, the pixel in the upper-left corner of the display corresponds to the 0 positions on both the horizontal and vertical axes, so it is usually said to be at x,y coordinates of 0,0.

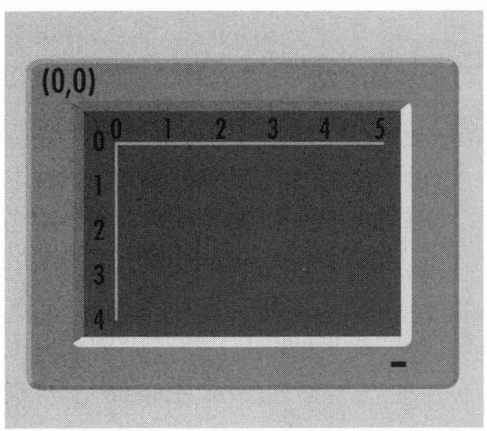


Figure 2-5 In this Cartesian coordinate system, the pixel is at 0,0

This position is referred to as the origin of the coordinate system, because it is the point at which the numbering systems on both of the axes begin. A pixel in the middle of the video display in mode 13h would have x,y coordinates of 160,100.

To change the color of a pixel at a given x,y coordinate location, we must first learn which offset within video memory corresponds to that location, then change the value stored at that address. Finding the offset within video memory that corresponds to a given coordinate pair is as easy as this instruction:

```
int offset = y * 320 + x;
```

After this instruction is executed, the *int* variable *offset* will be equal to the offset within video memory corresponding to the pixel at screen coordinates x and y. Because each line of pixels on the display consists of 320 pixels, *y* (which represents the number of lines from the top of the display to the line containing the pixel) must be multiplied by 320, then added to *x* (which represents the number of columns from the left side of the display to the column containing the pixel) to obtain this value.

We can set this pixel to the color white with the instruction

```
screen[offset] = 15;
```

It's assumed here that the pointer variable *screen* is pointing to the start of video memory and that the default palette is in use.

Once we know the offset of the pixel at a given pair of coordinates, it's simple to find the offset of the pixel to the immediate left or right of that pixel. Simply subtract 1 from or add 1 to the offset. Similarly, to find the offset of the pixel immediately below the pixel at a given offset, we add 320 to the offset, and to find the offset immediately above the pixel at a given offset, we subtract 320 from the offset.

Drawing a Line

And that brings us back to the problem of drawing a line. It's rare that we'll only need to change the color of individual pixels on the display. In depicting a wireframe maze, it's far more important that we be able to change the color of entire sequences of pixels to form lines. This isn't a trivial problem. In fact, it's one of the most complex problems that we'll be tackling in this book. (Don't let that frighten you away, though. Once you get through this discussion of computer line drawing, the rest of this book will be a snap!)

We'll need to make a distinction in this discussion between points and pixels. A point is a mathematical concept that can be defined by its coordinate position within a Cartesian plane. As you'll probably remember from grade school geometry class, a point has no width, depth, or breadth, and there are an infinite

number of them in a line or in a plane. In fact, there are an infinite number of points between any two other points on a line, no matter how close those points are to one another.

Pixels are the best approximation that we have for representing mathematical points on a video display, but there are some significant differences between pixels and points. Pixels *do* have width and breadth, and maybe even a little depth, but they're small enough that we can pretend that they don't. There are a finite number of pixels on the video display — 64,000 in mode 13h — and so there have to be a finite number of pixels in any line we draw on the display. It is possible, even somewhat trivial, to find two pixels on the video display that have no additional pixels between them.

Because there are an infinite number of points in a plane, and an infinite number of points between any other two points, the x,y coordinates of a point in a Cartesian coordinate system can have fractional values. It's perfectly legitimate to describe a point as being at coordinates 4.51232,7.177249 within a Cartesian plane, to cite but a single example. Pixels, on the other hand, can only have integral coordinates, without fractional values. To represent a point with fractional coordinates by a pixel with integer coordinates, we have to round the coordinates of the point to the nearest whole numbers to obtain the coordinates of the pixel that would best represent that point. The pixel most nearly equivalent to the point mentioned earlier in this paragraph would be the one at coordinates 5,7.

Endpoints

In geometry, a line is a sequence of points. However, to describe a line in terms of a Cartesian coordinate system, it's not necessary to specify the coordinates of every point that makes it up. (That wouldn't even be possible, since we'd have to specify an infinite number of coordinates.) A better method of describing a line, and the one that we'll be using in this chapter, is to specify the x,y coordinates of its endpoints — that is, the points at which it stops and starts. (Since lines don't necessarily have a direction, either end can be the starting point or the stopping point. In practice, the starting point of a line will be the point at which we start drawing it and the stopping point will be the point where we stop.) The line in Figure 2-6, for instance, starts at coordinates 2,1 and ends at coordinates 3,4 (or vice versa, depending on which end we start drawing it at). It's not necessary to specify the coordinates of any other points on the line because they can be calculated from these end points. I'll show you how to do that in a moment.

On a computer video display, a line is a sequence of pixels. We can describe a line on the video display in the same way that we describe a geometric line, by

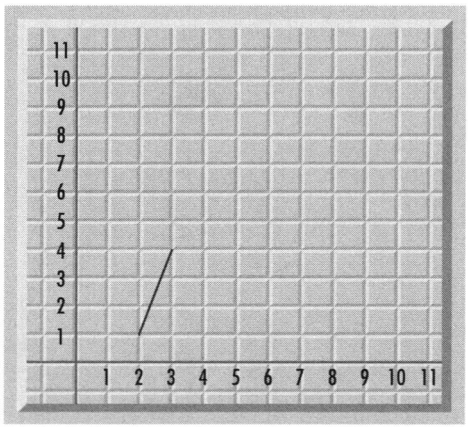


Figure 2-6 A line from coordinates 2,1 to coordinates 3,4

the coordinates of its endpoints. And we can draw the line as a sequence of pixels running from one endpoint to the other. But to do that, we'll need to figure out which pixels on the display come closest to representing some of the infinite number of points that make up the line. (Since pixels have width and breadth, we can imagine that each pixel is made up of an infinite number of points, so when we draw a line using a finite number of pixels, we are also drawing an infinite number of points.) This is no mean trick.

Horizontal and Vertical Lines

Some types of lines are easier to draw than others. The easiest of all are horizontal and vertical lines.

A horizontal line can be defined as a line that has the same y coordinates at both endpoints, but different x coordinates. The offset of every pixel in a horizontal line is 1 more than the offset of the pixel to its immediate left, and 1 less than the offset of the pixel to its immediate right. Similarly, the offset of every pixel in a vertical line is 320 more than the pixel above it, and 320 less than the pixel below it (see Figure 2-7). To draw a horizontal line, we first calculate the offset of the leftmost pixel in the line, change the color of that pixel, then add one repeatedly to the offset, changing colors as we go. To draw a vertical line, we calculate the offset of the topmost pixel, change its color, and add 320 repeatedly to the offset to get the offsets of the remaining pixels.

The program in Listing 2-6, for instance, draws a horizontal line across the middle of the mode 13h display, starting 10 pixels from the left side of the screen

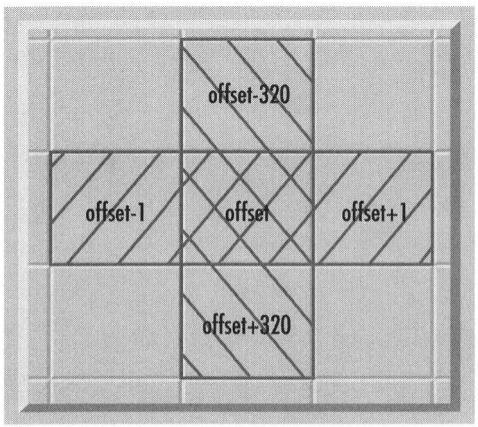
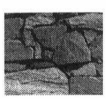


Figure 2-7 The offset of every pixel in a horizontal line is 1 more than the pixel to its left, and 1 less than the pixel to its right. The offset of every pixel in a vertical line is 320 more than the pixel above it, and 320 less than the pixel below it

and ending 10 pixels from the right. The setup is identical to that in the earlier WHITEOUT and PALETTE programs. The line drawing itself is done by a *for()* loop. Before this loop begins, the integer variables *startx* and *starty* are set to the x,y coordinates of the first pixel in the line. Then these coordinates are used to calculate the video memory offset of that pixel, using the formula that we introduced a few paragraphs ago. The *for()* loop simply steps the integer variable *i* through the 300 pixels on the line.



Listing 2-6 The HORZLINE program

```
//
// HORZLINE Version 1.0
//
// Draws a horizontal line across the mode 13h display
//
// Written by Christopher Lampton
// for Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  "screen.h"

void main(void)
{

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);
```

```
  // Create pointer to video memory:
  char far *screen=(char *)MK_FP(0xa000,0);

  // Set mode 13h:
  setmode(0x13);

  // Set starting coordinates for line:
  int startx=10;
  int starty=100;

  // Calculate video memory offset of coordinates:
  int offset=starty*320+startx;

  // Draw the line:
  for (int i=0; i<300; i++) screen[offset+i]=15;

  // Wait for user to press a key:
  while (!kbhit());

  // Reset original video mode:
  setmode(oldmode);
}
```

A vertical line can be drawn in pretty much the same manner, as shown in Listing 2-7. The major difference is that the value of the offset *i* must be incremented by 320 for each successive pixel in the line, moving the offset of the pixel a full screen line deeper into video memory.

Listing 2-7 The VERTLINE program

```
//
// VERTLINE Version 1.0
//
// Draws a vertical line down the mode 13h display
//
// Written by Christopher Lampton
// for Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  "screen.h"

void main(void)
{

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);
```

continued on next page

continued from previous page

```
  // Create pointer to video memory:
  char far *screen=(char *)MK_FP(0xa000,0);

  // Set mode 13h:
  setmode(0x13);

  // Set starting coordinates for line:
  int startx=160;
  int starty=10;

  // Calculate video memory offset of coordinates:
  int offset=starty*320+startx;

  // Draw the line:
  for(unsigned int i=0; i<57600; i+=320) screen[offset+i]=15;

  // Wait for user to press a key:
  while (!kbhit());

  // Reset original video mode:
  setmode(oldmode);
}
```

Lines of Arbitrary Slope

The real trick for a graphics programmer, though, is in writing a routine that can draw a line between any two arbitrary points on the video display. This takes some ingenuity. Fortunately, this ingenuity was demonstrated nearly three decades ago by a programmer named Bresenham, after whom the most popular line drawing algorithm is named. In this chapter we'll implement a version of Bresenham's Algorithm for the mode 13h display.

The key to drawing an arbitrary line is figuring out how much the position of each pixel on the line must change from the position of the pixel preceding it. In the horizontal and vertical line drawing programs we presented in the previous section, this was no problem at all: We changed the position of the pixel by one horizontal position for horizontal lines and one vertical position for vertical lines.

To figure out how much the position of each pixel changes in a line of arbitrary slope, we must first calculate the amount of change in each dimension. This isn't particularly difficult. It requires only that we calculate the line's *slope*. In mathematics the slope of a line is the change in the y coordinate from one end of a line to the other divided by the change in the x coordinate from one end to the other. For instance, the diagonal line in Figure 2-8 changes by 8 coordinates in the x position and 8 coordinates in the y position, from the left end of the line to the right. Thus we say that this line has a slope of 8/8 or 1. This tells us how much the y coordinate changes from one pixel to the next along the line, if we increment the x coordinate by 1 on each pixel. In this case it changes by 1. Invert

the slope — flip it top to bottom — and it tells us how much the x coordinate would need to change from one end of the line to the other while the y coordinate is being incremented by 1. Since 8/8 is the same inverted, this change is also 1. So to draw this diagonal line, we would need to change the pixel coordinates by 1 in the y dimension while incrementing the x coordinate, or by 1 in the x dimension while incrementing the y coordinate.

Unfortunately, not all line slopes are integral. Most contain fractions, meaning that the coordinates of the pixel must be changed by a fractional amount in either the x or y dimensions. For instance, the line in Figure 2-9 has a slope of 1/2, because it changes by twice as much in the horizontal dimension over the length of the line as in the vertical dimension. This tells us that the y coordinate should be incremented by 1/2 for each pixel, while the x coordinate is being incremented by 1. Unfortunately, there's no such thing as half a pixel. We could draw the same line by flipping the slope over and incrementing the x coordinate by 2 while incrementing the y coordinate by 1, but then the line would have holes in it, because we'd be skipping pixels.

The only solution that works for pixel-oriented video displays of the sort produced by the VGA adapter is to ignore fractional changes in pixel position until they've added up to at least 1, then advance the pixel to the next coordinate in that position. For instance, the vertical position of the pixels in the line in Figure 2-10 may only need to be incremented by 1/2 a coordinate every time the horizontal position is incremented by 1, but in real video terms it needs to be incremented by a full coordinate every time the horizontal position is incremented by 2. So we can draw this line if we increment the y position of the

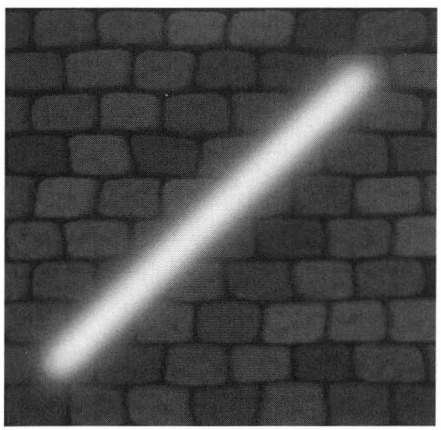


Figure 2-8 A diagonal line

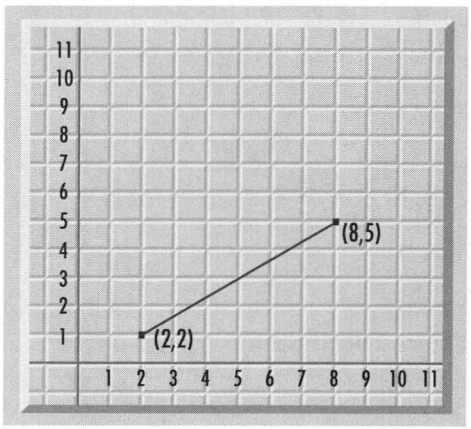


Figure 2-9 A line with a slope of 1/2

pixels once every *other* time we increment the x position of the pixels, as shown in Figure 2-10.

In fact, we can use this method to draw lines with any slope at all. The simplest way to implement this system is to maintain fractional values for the x,y coordinates of the pixels, but draw the pixels on the display using a pixel drawing function that truncates the fractional portion of the coordinate values to the nearest integer when calculating the video memory offsets of the pixels. This will cause the pixel to automatically advance to the next position when the fractional value increases to the next integer — when a couple of 1/2 increments add up to an increment of 1, for instance. A simple line-drawing function that would draw any line with a slope from 0 to 1 (that is, from horizontal to diagonal) could be written by calculating the slope, incrementing the x coordinate by 1 for every pixel of the line, and adding the slope to the y coordinate for every pixel. To draw a line with a slope greater than 1 (that is, from diagonal to vertical), we would simply invert the slope, increment the y coordinate by 1 for each pixel, and add the inverted slope to the x coordinate. We could then call a pixel-drawing function to draw each pixel as its position is calculated.

But we don't want to deal with fractions in our line-drawing routine. I'll talk more about the reasons for this when we discuss optimization, but the basic reason is that fractions are slow. The line-drawing function that we develop in this chapter should avoid fractions at all cost, using only integer math. The trick to avoiding fractions is to avoid division — that is, not to calculate the slope —

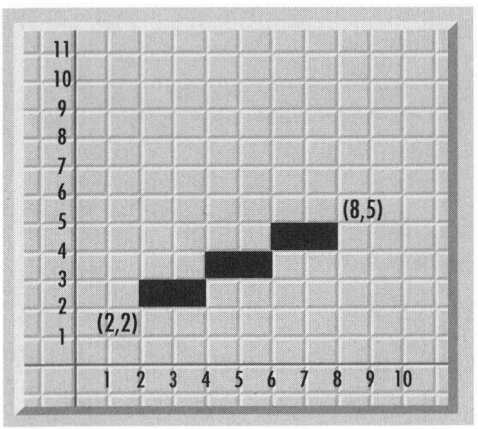


Figure 2-10 The line in Figure 2-9 drawn by incrementing the y pixel position once for every time that the x pixel position is incremented twice

because it is this division that produces the fractions in the first place. But how can we tell how much to advance the pixel position if we don't calculate the slope?

Bresenham's Algorithm

Bear in mind that division is really nothing more than repeated subtraction. If we repeatedly subtract one number from another number until the remaining value is less than the number that we are subtracting, then we have effectively divided the second number by the first. The number of repetitions we need to subtract the first number from the second is the quotient of the division, and the amount left over is the remainder. For instance, we can subtract 2 five times from 11, leaving a result of 1 — which is smaller than 2. Thus 11 divided by 2 is 5, with a remainder of 1.

Similarly, if we add a number repeatedly to itself until it is larger than a second number, then the number of additions that we have performed is the quotient of the first number divided into the second, plus 1. If we then subtract the second number from the first, we get the remainder.

This process, which we might call incremental division, is the secret to implementing Bresenham's algorithm. First we calculate the change in the x and y coordinates along the length of the line. (This is simply a matter of subtracting the beginning x and y coordinates from the ending x and y coordinates and taking the absolute values of the results.) Call these two values *xdiff* and *ydiff*. If the slope is less than 1 — that is, if *xdiff* is greater than *ydiff* — we increment the x coordinate by 1 after drawing each pixel. To determine when to increment the y coordinate, we create a value called the *error term*, which is initially set to 0. We call it the error term because it will tell us how far our pixelated line is deviating from a geometric line. After drawing each pixel, we add *ydiff* to the error term and check to see if the value of the error term is greater than *xdiff*. If it isn't, we leave the y coordinate as it is. If it is, then our y coordinate has deviated by (at least) a full pixel position from the actual geometric position of the corresponding point on the line, so we increment the y coordinate and subtract *xdiff* from the error term.

To see how this works, imagine that *xdiff* is 42 and *ydiff* is 20. The slope of the line would be 20/42 or 10/21. We would need to increment the y coordinate 10 times for every time that we increment the x coordinate 21 times. The error term helps us do this. Adding *ydiff* to the initial error term value of 0 gives us a value of 20. We compare this to *xdiff* and see that the value of the error term has not yet exceeded the value of *xdiff*, so we don't advance the y coordinate. The next time we add *ydiff* to the error term, the result is a value of 40, which still doesn't exceed the value of *xdiff*, so we leave the y coordinate untouched once

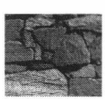
again. But the third time the value of the error term becomes 60, which does exceed the value of *xdiff*, so we increment the y coordinate. This means that the first three pixels of the line are drawn at the same y coordinate, but the y coordinate of the fourth pixel is advanced by 1. We then subtract the value of *xdiff* from the error term, producing an error term value of 18. Because the error term now starts out at a higher value, we will only be able to add *ydiff* to it twice before it exceeds the value of *xdiff*. As a result, the y coordinate is incremented after only two more pixels. This will continue to be the case for the next 19 pixels, at which point the initial value of the error term will have returned to 0. The process then repeats, with three pixels in a row having the same y coordinate, then only two pixels in a row for the next 20 repetitions of the process. The result? The y coordinate will be incremented 10 times for every 21 times that the x coordinate will be incremented, just as if we had calculated the slope.

If the slope is greater than 1 — that is, if *ydiff* is larger than *xdiff* — then we must do this the other way around. We increment the y coordinate by 1 for each pixel on the line, adding *xdiff* to the error term after each pixel is drawn and incrementing the y coordinate when the error term becomes greater than *ydiff*. We also must make allowance for the direction in which the line is moving — that is, whether it's going from a small coordinate to a large coordinate or vice versa in the x and y dimensions. When going from a small coordinate to a large coordinate — we'll call this the positive direction — we must increment the pixel coordinates in that dimension; but if we're going from a large coordinate to a small one (the negative direction), we must decrement them.

Listing 2-8 shows how all of this is done in an actual C++ implementation of Bresenham's algorithm, which is included on the disk in the file BRESN.CPP. To use this function, you must include the BRESN.H header in your programs. It can be called using this syntax:

```
linedraw(x1,y1,x2,y2,color,screen);
```

The first four parameters are the starting and ending coordinates of the line. The *color* parameter is the number of the palette color in which the line is to be drawn and *screen* is a pointer to the video display (or to an offscreen buffer).



Listing 2-8 The linedraw() function

```
void linedraw(int x1,int y1,int x2,int y2,int color,
              char far *screen)
{
  int y_unit,x_unit;        // Variables for amount of change
                            //  in x and y
```

```
int offset=y1*320+x1; // Calculate offset into video RAM

int ydiff=y2-y1;      // Calculate difference between
                      //  y coordinates
if (ydiff<0) {        // If the line moves in the negative
                      //  direction
  ydiff=-ydiff;       // ...get absolute value of difference
  y_unit=-320;        // ...and set negative unit in
                      //  y dimension
}
else y_unit=320;      // Else set positive unit in
                      //  y dimension

int xdiff=x2-x1;      // Calculate difference between
                      //  x coordinates
if (xdiff<0) {        // If the line moves in the
                      //  negative direction
     xdiff=-xdiff;    // ...get absolute value of
                      //  difference
     x_unit=-1;       // ...and set negative unit
                      // in x dimension
}
else x_unit=1;        // Else set positive unit in
                      //  y dimension

int error_term=0;     // Initialize error term
if (xdiff>ydiff) {    // If difference is bigger in
                      //  x dimension
     int length=xdiff+1;          // ...prepare to count off in
                                  //  x direction
  for (int i=0; i<length; i++) {  // Loop through points
                                  //  in x direction
    screen[offset]=color;         // Set the next pixel in the
                                  //  line to COLOR
    offset+=x_unit;               // Move offset to next pixel
                                  //  in x direction
    error_term+=ydiff;            // Check to see if move
                                  // required in y direction
    if (error_term>xdiff) {       // If so...
      error_term-=xdiff;          // ...reset error term
      offset+=y_unit;             // ...and move offset to next
                                  //  pixel in y dir.
         }
     }
}
else {                            // If difference is bigger in
                                  //  y dimension
  int length=ydiff+1;             // ...prepare to count off in
                                  //  y direction
  for (int i=0; i<length; i++) {  // Loop through points
                                  //  in y direction
    screen[offset]=color;         // Set the next pixel in the
```

continued on next page

continued from previous page

```
                                    //  line to COLOR
      offset+=y_unit;               // Move offset to next pixel
                                    //  in y direction
      error_term+=xdiff;            // Check to see if move
                                    //  required in x direction
      if (error_term>0) {           // If so...
        error_term-=ydiff;          // ...reset error term
        offset+=x_unit;             // ...and move offset to next
                                    //  pixel in x dir.
      }
    }
  }
}
```

This function divides neatly into two parts, one for lines with slopes less than 1 (where *xdiff* is greater than *ydiff*), and one for lines with slopes greater than 1 (where *ydiff* is greater than *xdiff*). (Lines with a slope of precisely 1 can be handled by either part.) The variable *x_unit* is set to either positive or negative 1, depending on whether the line is moving in the positive or negative direction (that is, from lower to higher *x* or vice versa). Similarly, the variable *y_unit* is set to positive or negative 320. These variables are then added to the pixel positions to increment them from one position to the next in the corresponding dimensions.

The variable *error_term* is initialized to 0 and either *ydiff* (for lines with slope less than 1) or *xdiff* (for lines with slope greater than 1) is added to it repeatedly, until it exceeds the value of either *xdiff* or *ydiff* (whichever is not being added to it for the current slope). Then the pixel is incremented in the appropriate dimension, *ydiff* (or *xdiff*) is subtracted from *error_term* and the process begins again, until the end of the line is reached. A *for()* loop is used to increment the pixel coordinate in either the x or y dimension from one end of the line to the other.

At Last ... the Maze

We'll use this implementation of Bresenham's algorithm to draw the "wires" in our wireframe maze. To create a wireframe maze, we must do two things: Design a data structure for storing the maze in the computer's memory, and design an algorithm for drawing the maze on the screen.

The first of these is the easier problem to solve. If we restrict ourselves to mazes that can be depicted on a piece of graph paper using horizontal and vertical lines that correspond to the graph lines, then we can store a description of a maze as a two-dimensional array of bytes (that is, as a two-dimensional *char* array). Consider, by way of example, the maze in Figure 2-11, which has been drawn on a piece of graph paper. Each square on the graph paper represents a single *position* in the maze. Squares that have been shaded are *wall squares* — that

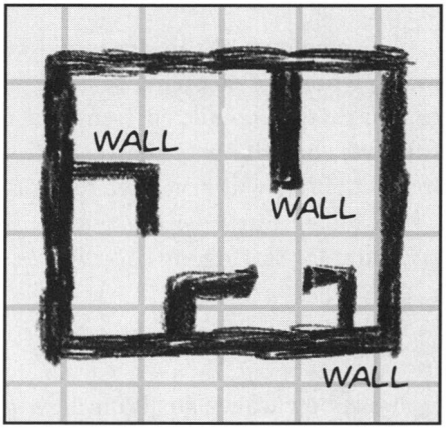


Figure 2-11 A maze drawn on a sheet of graph paper

is, impassable squares in which solid cement extends from the floor to the ceiling. An adventurer wandering through this maze would not be able to enter these squares. Unshaded squares are empty squares, in which an adventurer can travel.

Every square in this maze can be given a coordinate position, similar to the way in which we give coordinate positions to pixels on the video display. If we place the origin of the maze coordinate system at the upper-left corner, then the square in that corner would have x,y coordinates of 0,0. The square to its immediate right would have x,y coordinates of 1,0. The square immediately below it would have x,y coordinates of 0,1. And so forth.

We can store such a maze in the computer as a 16 by 16 array of *char*, like this:

```
char maze[16][16]={
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,1,0,1,1,1,0,0,0,0,0,1,1,0,1},
  {1,0,1,0,1,0,1,0,1,1,1,0,1,1,0,1},
  {1,0,1,0,1,0,1,0,0,0,1,0,0,1,0,1},
  {1,0,1,1,1,0,1,0,1,0,1,1,1,1,0,1},
  {1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1},
  {1,0,1,1,1,0,1,0,1,1,1,0,1,1,0,1},
  {1,0,1,0,0,0,1,1,0,0,1,0,1,1,0,1},
  {1,0,1,0,1,0,1,1,1,1,1,0,1,1,0,1},
  {1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,1,1,1,0,1,0,1,0,1,1,1,1,0,1},
  {1,0,0,0,1,0,1,0,1,0,0,0,0,1,0,1},
  {1,0,1,1,1,0,1,1,1,1,1,1,1,1,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
};
```

The elements of this array that are set to 1 represent the shaded squares in the maze, while the elements that are set to 0 represent open squares. (It may occur to you that we could represent this maze using only a bit per square rather than a full byte, but the byte representation will prove useful in the more sophisticated mazes that we will develop later in this book.)

To locate our position within the maze, it would be useful to have a data structure for storing x,y coordinates. We'll define such a structure like this:

```
struct xy {
  int x,y;
};
```

Now, how do we go about drawing the maze on the display?

One Square at a Time

Imagine yourself standing in the middle of a maze, looking into the distance. What would you see? Probably, you'd see something like the wireframe maze representation in Figure 2-12 (albeit with the addition of such real world frills as colors and textures). A long hallway stretches out in front of you, interrupted periodically by doors opening to both sides.

To draw such a representation on the video display, we need to locate the viewer's position and orientation within the maze, then work our way down the

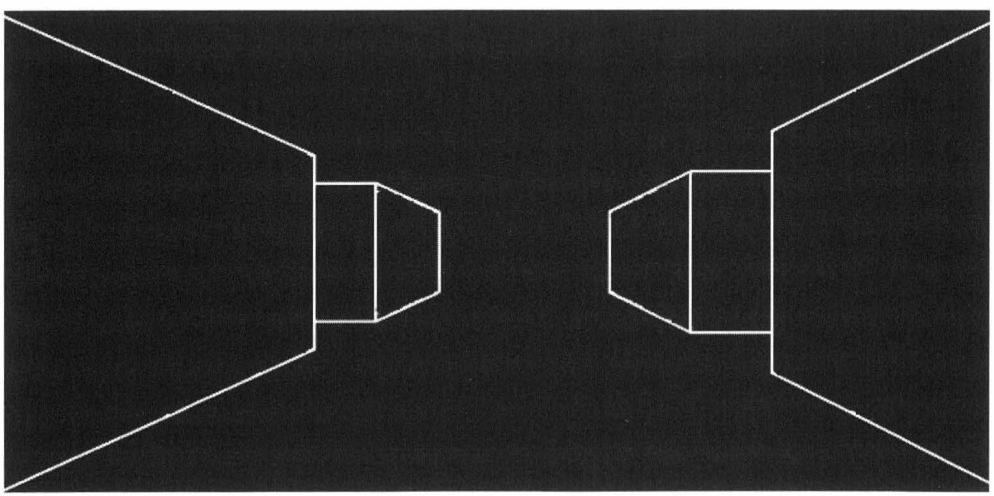


Figure 2-12 A wireframe representation of a maze as seen by someone standing in the middle of it

hallway along which he or she is currently looking. Where we find walls, we should draw walls. Where we find openings in the walls, we should draw openings.

First, we need a variable to store the viewer's current position. We'll use the *xy* structure that we defined a moment ago to declare a variable called *pos:*

```
struct xy pos={1,3};
```

This indicates that the viewer is standing at coordinate position 1,3 within the maze, one square from the left edge and three squares from the right edge.

We'll also need a way to define the viewer's orientation — that is, whether the viewer is facing in the positive x direction (toward higher x coordinate positions in the maze), the negative x direction (toward lower x coordinates in the maze), the positive y direction, or the negative y direction. We'll arbitrarily refer to the positive x direction as direction 0, the positive y direction as direction 1, the negative x direction as direction 2, and the negative y direction as direction 3 (see Figure 2-13). That way, we can store the direction as an integer variable:

```
int direction=1;
```

But how will we tell the computer what these directions mean? We'll create an array of type *xy* which will hold the increments for each of these directions — that is, an *x* value of 1 for the positive x direction, an *x* value of -1 for the negative x direction, and so forth. We'll call this array *increment* and we'll declare it like this:

```
struct xy increment[4]={{-1,0},{0,1},{1,0},{0,-1}};
```

There is one element of this array for each of the four positions in which the viewer can face. Restricting the viewer's orientation to these four positions

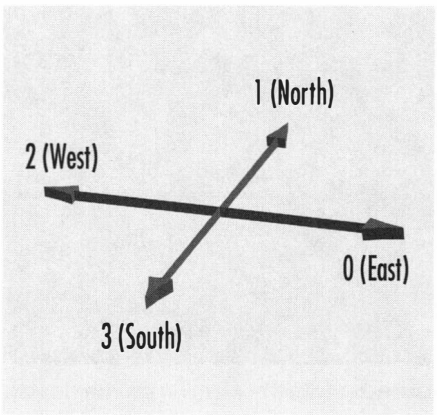


Figure 2-13 The four compass directions

simplifies the problem of drawing the maze, since we only have to draw it from these four positions. (Most of the early maze games worked this way.) Later, we'll learn how to draw a maze from any arbitrary orientation.

For reasons that we'll see in a moment, we'll also need a pair of arrays to give us the increments along the x and y axes for paths through the maze that move to the left and right of the viewer's current orientation. We'll call these arrays *left* and *right:*

```
struct xy left[4]={{0,-1},{-1,0},{0,1},{1,0}};
struct xy right[4]={{0,1},{1,0},{0,-1},{-1,0}};
```

We could draw the image of the maze to fill the entire display, but we'll rarely, if ever, need to do that in practice. Usually, we'll be drawing the maze in a window filling part of the display and placing other information in the remaining space. So let's draw a small box on the display to delineate the window in which the maze will be drawn:

```
void drawbox(char *screen)
{
  linedraw(82,19,294,19,15,screen);
  linedraw(294,19,294,119,15,screen);
  linedraw(294,119,82,119,15,screen);
  linedraw(82,119,82,19,15,screen);
}
```

This short function, called *drawbox(),* uses the *linedraw()* function that we developed earlier to draw a box on the display with its upper-left corner at coordinates 82,19 and its lower-right corner at coordinates 294,119 (see Figure 2-14).

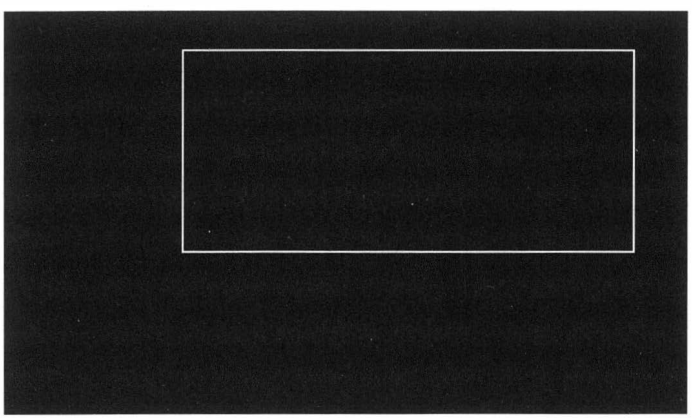


Figure 2-14 The box drawn on the display by the DRAWBOX() function

To draw the maze itself, we need to know how many squares the viewer is able to see into the distance. We'll arbitrarily choose a value of 4, which we'll store in the integer variable *visibility:*

```
int visibility=4;
```

Now we'll need to draw each of the four squares visible along the viewer's line of sight. We'll step through all four squares using a *for()* loop:

```
for(int dist=0; dist<visibility; dist++) {
```

We'll use a separate drawing routine for each of these four squares, to take the changing perspective into account. (We could also use a single drawing routine in a loop and calculate the differences in perspective on the fly, but for reasons of clarity we'll go with the former approach.) First we need to calculate the coordinates of the square, or block, in which the viewer is currently standing and the coordinates of the blocks to the immediate left and right of that block. We'll do this with the aid of the *left*, *right*, and *increment* arrays:

```
// Find current square of maze:

block.x=pos.x+dist*increment[direction].x;
block.y=pos.y+dist*increment[direction].y;

// Find square to the left of current square:

lblock.x=block.x+left[direction].x;
lblock.y=block.y+left[direction].y;

// Find square to the right of current square:

rblock.x=block.x+right[direction].x;
rblock.y=block.y+right[direction].y;
```

We'll use a *switch* statement to choose among the four different drawing functions:

```
switch(dist) {
```

For the current square, which is treated by the program as the square at a distance of 0 from the viewer's position, we'll check to see if the square to the left is open or closed. If closed, we'll draw a wall on the left side of the view window. If open, we'll draw a door:

```
if (maze[lblock.x][lblock.y]) {
  linedraw(82,19,135,44,15,screen);
  linedraw(135,44,135,93,15,screen);
  linedraw(135,93,82,118,15,screen);
}
```

```
else { // Else draw opening
  linedraw(82,44,135,44,15,screen);
  linedraw(135,44,135,93,15,screen);
  linedraw(135,93,82,93,15,screen);
}
```

Then we'll do the same thing with the square on the right:

```
// Is wall open to the right?
// If not, draw wall
if (maze[rblock.x][rblock.y]) {
  linedraw(294,19,242,44,15,screen);
  linedraw(242,44,242,93,15,screen);
  linedraw(294,118,242,93,15,screen);
}
else { // Else draw opening
  linedraw(294,44,242,44,15,screen);
  linedraw(242,44,242,93,15,screen);
  linedraw(242,93,294,93,15,screen);
}
```

For the next square — the square at a distance of 1 in the viewing direction — we first need to check to see if the square itself is open or closed. If closed, we draw a wall in the middle of the viewing window:

```
// Can we see the next square?
// If not, draw wall
if (maze[block.x][block.y]) {
  linedraw(135,44,135,93,15,screen);
     linedraw(242,44,242,93,15,screen);
     linedraw(135,44,242,44,15,screen);
     linedraw(135,93,242,93,15,screen);
   }
```

If it's open, then we do the same thing with the next square as we did with the current one: Draw a wall on each side if the left and/or right squares are closed or an opening if those squares are open, with the coordinates adjusted for perspective:

```
   else { // Else draw sides of next square
     if (maze[lblock.x][lblock.y]) {
       linedraw(135,44,162,57,15,screen);
       linedraw(162,57,162,80,15,screen);
       linedraw(162,80,135,93,15,screen);
     }
     else {
       linedraw(135,57,162,57,15,screen);
       linedraw(162,57,162,80,15,screen);
       linedraw(162,80,135,80,15,screen);
     }
```

```
    if (maze[rblock.x][rblock.y]) {
      linedraw(242,44,215,57,15,screen);
      linedraw(215,57,215,80,15,screen);
      linedraw(215,80,242,93,15,screen);
    }
    else {
      linedraw(242,57,215,57,15,screen);
      linedraw(215,57,215,80,15,screen);
      linedraw(215,80,242,80,15,screen);
    }
  }
```

We then repeat this process for the two remaining visible squares, adjusting the line coordinates again for perspective. If the view of any of these squares is blocked by a wall, we'll want to short circuit the drawing of the maze, so we place a statement at the end of the loop to watch for this situation and execute a *break* instruction if it is encountered:

```
if (maze[block.x][block.y]) break;
```

What happens if the viewer is standing within four squares of the edge of the maze and is looking toward the edge? How do we keep from extending our line of sight off the edge of the maze array? If you go back and look at the initialization statement for the maze array, you'll see that the maze is entirely bordered by walls; it's not possible to see past the edge. If you should alter the maze array so that there are holes in this wall, what would you see through them? Random pieces of maze, probably, generated by junk in the computer's memory.

There's one situation that this maze-drawing program can't handle. If four maze squares arranged in a square pattern relative to one another are left empty, there will be odd-looking gaps in the drawing. (This would be equivalent to two consecutive openings off the same side of the hallway leading away in the viewer's line of sight.) Just as we've designed the maze so that it's impossible to see past the edge, we've also designed it so that four empty squares never appear together arranged in a square. Later, we'll write maze-drawing routines that *can* handle this situation.

The complete maze-drawing program is shown in Listing 2-9.

Listing 2-9 The MAZE.CPP program

```
// MAZE.CPP Version 1.0
// Draw a single view inside a 3D wireframe maze
//
// Written by Christopher Lampton
//   for Gardens of Imagination (Waite Group Press)
```

continued on next page

continued from previous page

```
#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  "screen.h"
#include  "bresn.h"

char maze[16][16]={
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,1,0,1,1,1,0,0,0,0,0,1,1,0,1},
  {1,0,1,0,1,0,1,0,1,1,1,0,1,1,0,1},
  {1,0,1,0,1,0,1,0,0,0,1,0,0,1,0,1},
  {1,0,1,1,1,0,1,0,1,0,1,1,1,1,0,1},
  {1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1},
  {1,0,1,1,1,0,1,0,1,1,1,0,1,1,0,1},
  {1,0,1,0,0,0,1,1,0,0,1,0,1,1,0,1},
  {1,0,1,0,1,0,1,1,1,1,1,0,1,1,0,1},
  {1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,1,1,1,0,1,0,1,0,1,1,1,1,0,1},
  {1,0,0,0,1,0,1,0,1,0,0,0,0,1,0,1},
  {1,0,1,1,1,0,1,1,1,1,1,1,1,1,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
};

typedef struct xy {
  int x,y;
};

struct xy increment[4]={{-1,0},{0,1},{1,0},{0,-1}};
struct xy left[4]={{0,-1},{-1,0},{0,1},{1,0}};
struct xy right[4]={{0,1},{1,0},{0,-1},{-1,0}};
struct xy pos={1,3};

int direction=1;
int visibility=4;

void drawmaze(char *);
void drawbox(char *);

void main(void)
{

  // Create pointer to video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Put display in mode 13h:
```

```
  setmode(0x13);

  // Clear display:
  cls(screen);

  // Draw window on display:
  drawbox(screen);

  // Draw the maze:
  drawmaze(screen);

  // Wait for user to hit a key:
  while (!kbhit());

  // Restore old video mode:
  setmode(oldmode);
}

void drawbox(char *screen)
{
  linedraw(82,19,294,19,15,screen);
  linedraw(294,19,294,119,15,screen);
  linedraw(294,119,82,119,15,screen);
  linedraw(82,119,82,19,15,screen);
}

void drawmaze(char *screen)

// Draw the maze stored in array maze[] into video
// memory at address screen.

{
  struct xy block,lblock,rblock;
  int oldleft,oldright;

  // Draw the maze at each distance allowed by visibility
  for(int dist=0; dist<visibility; dist++) {

    // Find current square of maze:
    block.x=pos.x+dist*increment[direction].x;
    block.y=pos.y+dist*increment[direction].y;

    // Find square to the left of current square:
    lblock.x=block.x+left[direction].x;
    lblock.y=block.y+left[direction].y;

    // Find square to the right of current square:
    rblock.x=block.x+right[direction].x;
    rblock.y=block.y+right[direction].y;

    // Draw image of squares according to distance:
```

continued on next page

continued from previous page

```
    switch(dist) {
      case 0:  // Draw current square

        // Is wall open to the left?
        // If not, draw wall
        if (maze[lblock.x][lblock.y]) {
          linedraw(82,19,135,44,15,screen);
          linedraw(135,44,135,93,15,screen);
          linedraw(135,93,82,118,15,screen);
        }
        else { // Else draw opening
          linedraw(82,44,135,44,15,screen);
          linedraw(135,44,135,93,15,screen);
          linedraw(135,93,82,93,15,screen);
        }

        // Is wall open to the right?
        // If not, draw wall
        if (maze[rblock.x][rblock.y]) {
          linedraw(294,19,242,44,15,screen);
          linedraw(242,44,242,93,15,screen);
          linedraw(294,118,242,93,15,screen);
        }
        else { // Else draw opening
          linedraw(294,44,242,44,15,screen);
          linedraw(242,44,242,93,15,screen);
          linedraw(242,93,294,93,15,screen);
        }
        break;
      case 1: // Repeat for next square

        // Can we see the next square?
        // If not, draw wall
        if (maze[block.x][block.y]) {
          linedraw(135,44,135,93,15,screen);
          linedraw(242,44,242,93,15,screen);
          linedraw(135,44,242,44,15,screen);
          linedraw(135,93,242,93,15,screen);
        }
        else { // Else draw sides of next square
          if (maze[lblock.x][lblock.y]) {
            linedraw(135,44,162,57,15,screen);
            linedraw(162,57,162,80,15,screen);
            linedraw(162,80,135,93,15,screen);
          }
          else {
            linedraw(135,57,162,57,15,screen);
            linedraw(162,57,162,80,15,screen);
            linedraw(162,80,135,80,15,screen);
          }
          if (maze[rblock.x][rblock.y]) {
```

```
      linedraw(242,44,215,57,15,screen);
      linedraw(215,57,215,80,15,screen);
      linedraw(215,80,242,93,15,screen);
    }
    else {
      linedraw(242,57,215,57,15,screen);
      linedraw(215,57,215,80,15,screen);
      linedraw(215,80,242,80,15,screen);
    }
  }
  break;
case 2: // Do it again
  if (maze[block.x][block.y]) {
    linedraw(162,57,162,80,15,screen);
    linedraw(215,57,215,80,15,screen);
    linedraw(162,57,215,57,15,screen);
    linedraw(162,80,215,80,15,screen);
  }
  else {
    if (maze[lblock.x][lblock.y]) {
      linedraw(162,57,175,63,15,screen);
      linedraw(175,63,175,74,15,screen);
      linedraw(175,74,162,80,15,screen);
    }
    else {
      linedraw(162,63,175,63,15,screen);
      linedraw(175,63,175,74,15,screen);
      linedraw(175,74,162,74,15,screen);
    }
    if (maze[rblock.x][rblock.y]) {
      linedraw(215,57,202,63,15,screen);
      linedraw(202,63,202,74,15,screen);
      linedraw(202,74,215,80,15,screen);
    }
    else {
      linedraw(215,63,202,63,15,screen);
      linedraw(202,63,202,74,15,screen);
      linedraw(202,74,215,74,15,screen);
    }
  }
  break;
case 3: // And again
  if (maze[block.x][block.y]) {
    linedraw(175,63,175,74,15,screen);
    linedraw(202,63,202,74,15,screen);
    linedraw(175,63,202,63,15,screen);
    linedraw(175,74,202,74,15,screen);
  }
  else {
    if (maze[lblock.x][lblock.y]) {
      linedraw(175,63,182,66,15,screen);
```

continued on next page

continued from previous page

```
            linedraw(182,66,182,70,15,screen);
            linedraw(182,70,175,74,15,screen);
          }
          else {
            linedraw(175,66,182,66,15,screen);
            linedraw(182,66,182,70,15,screen);
            linedraw(182,70,175,70,15,screen);
          }
          if (maze[rblock.x][rblock.y]) {
            linedraw(202,63,195,66,15,screen);
            linedraw(195,66,195,70,15,screen);
            linedraw(195,70,202,74,15,screen);
          }
          else {
            linedraw(202,66,195,66,15,screen);
            linedraw(195,66,195,70,15,screen);
            linedraw(195,70,202,70,15,screen);
          }
        }
        break;
    }

    // If view is obscured by wall, stop drawing:

    if (maze[block.x][block.y]) break;
  }
}
```

Run it and — *voila!* — a three-dimensional wireframe view of a maze will be drawn on the computer display (see Figure 2-15). Change the viewer position and orientation (in the variables *pos* and *direction*) and recompile to see the maze from other positions. (Be careful about putting the viewer inside a closed square, however, since the program never checks for this possibility.)

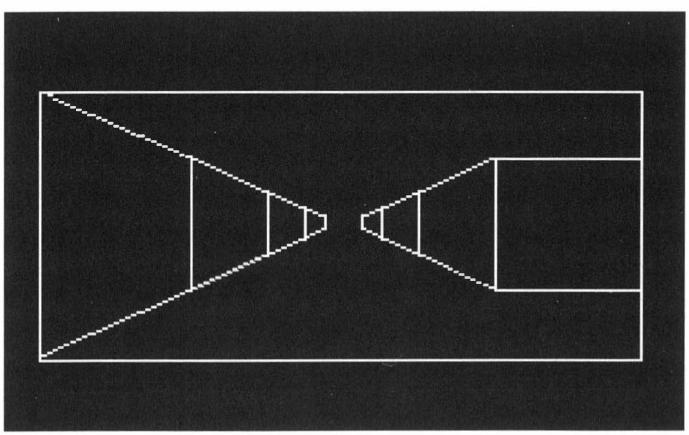
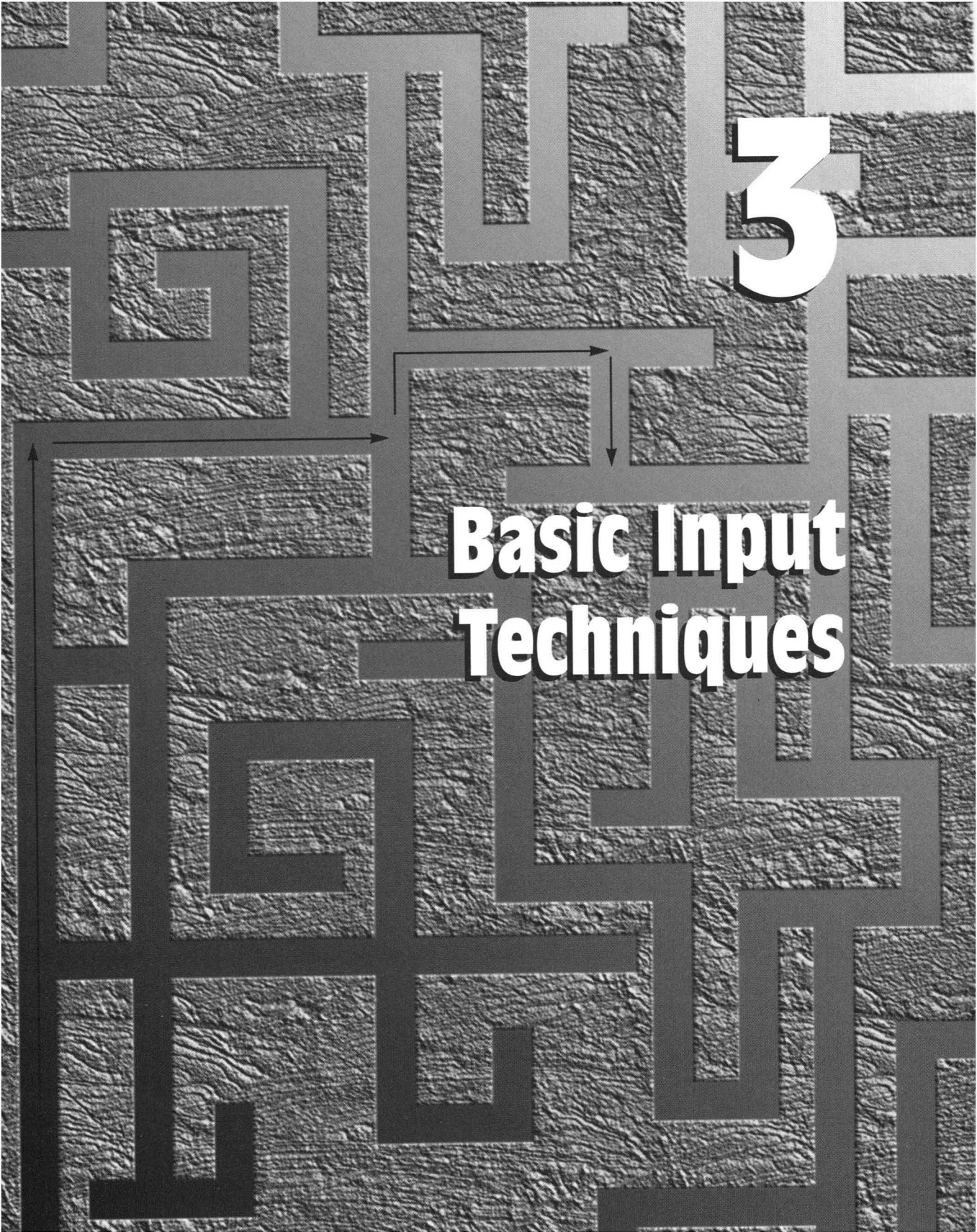


Figure 2-15 The three-dimensional maze view drawn by the MAZE.CPP program

While it's all well and good to draw an image of a maze as it appears from a single position, as we have done here, a true maze program should be animated. The user should be able to explore the maze rather than sit in a single spot inside it. In the next chapter we'll look at ways in which the user can interact with our maze-drawing code — and then we'll create a version of our wireframe maze-drawing program that will allow the viewer to explore an animated maze using the keyboard, mouse, or joystick.



3

Basic Input Techniques

raphics are an important part of what makes a computer game function, but they aren't the only part. You also need a way to control or interact with the game, so that you can move characters around, fly spaceships, bash evil wizards, shoot monsters, and otherwise take out your aggressive instincts on innocent pixels. If there were no way to interact with the game, you might as well turn off the computer and turn on the TV.

The maze-drawing code that we developed in the last chapter could be used as the core of a computer game in the mold of the early Wizardry CRPGs. But to make that game interactive, we need to give the player a way to communicate with the computer.

There are three major input devices used in computer games. Not coincidentally, two of these are also the major input devices used in all PC applications: the keyboard and the mouse. The third, the analog joystick, is used almost exclusively for games. Let's look at each of these devices in turn, starting with the keyboard.

The PC Keyboard

The keyboard is the most familiar of all input devices, for the simple reason that there's one attached to every computer. Only some users of the games that we

develop in this book will have joysticks. Quite a few will have mice. But each and every one will have a keyboard. That makes the keyboard the most important input device you can learn to program. It won't necessarily be the *best* device for the task at hand; in many cases, a game will be more easily controlled via the joystick or the mouse. But it's pretty much obligatory for game programmers to support the ubiquitous keyboard, because it's the one input device you can be sure that your users will have.

That may sound like good news. Surely the keyboard is easier to program than such relatively esoteric devices as joysticks and mice. The libraries that came with your C/C++ compiler are filled with functions for working with the keyboard. You can use these functions to cobble together a quick and dirty keyboard interface, then turn to the real work of programming the mouse and the joystick. You could, for instance, use the *kbhit()* function to determine if a key has been pressed and the *getch()* function to learn which key it was. Alternatively, you could call the INT 16h functions from the ROM BIOS if you need an even lower- level method of monitoring the keyboard.

Alas, things aren't quite that simple. The PC keyboard is a strange and complicated thing. It can do some nifty tricks for the programmer who knows how to make it strut its stuff, but the keyboard functions that are supplied with most compilers don't support the type of keyboard input that most games require. Neither do the keyboard routines built into the PC's ROM BIOS. In fact, if we're going to provide proper keyboard input for our maze games, we really have no choice but to write our own low-level machine language input functions.

Two Keys at a Time

Perhaps the single greatest deficiency of the keyboard routines that come bundled with the compiler and the ones that are built into the ROM BIOS is that they don't allow us to detect when two keys are being pressed at the same time. For most nongame applications (with the exception of terminate-and-stay-resident utilities, which are a whole different ball of wax), this isn't particularly important. Programmers developing word processors, for instance, usually only care about the order in which keys are pressed, so detecting the simultaneous pressing of two keys is largely irrelevant to them.

Game programmers, by contrast, are in constant need of this information. Suppose, for instance, that we want to allow the user of our maze games to move forward through the maze by pressing the ↑ key while firing bursts from a laser gun by pressing the SPACEBAR. If we could only detect the pressing of one key at a time, it would be necessary for the player to stop moving every time he or she needed to fire the weapon, a requirement that would hardly endear most players to our game.

Luckily for us, it's quite possible to detect the simultaneous pressing of two keys on the PC keyboard — if we write our own low-level keyboard code. But low-level keyboard code is notoriously tricky to write, because it involves entering the shadowy and mysterious world of interrupt service programming. If you're the sort of programmer who can write three UNIX-compatible operating systems before breakfast, this will be old stuff to you. The rest of you, however, should sit up and pay close attention.

The Keyboard Interrupt

The secret of programming the PC keyboard is realizing that the keyboard is a computer in its own right. More precisely, it contains within it a special purpose microprocessor called an Intel 8042 or 8048, depending on what generation of PC you happen to be programming. (Because the 8042 or a compatible processor is used in the keyboards of later PCs, we'll use that number to refer to the keyboard processor in this text.) The 8042 monitors the pressing of keys on the keyboard and sends an electronic report back to the computer's main CPU every time a key is pressed. (It also sends back a report when a key is released, which allows us to detect the simultaneous pressing of two keys. More about that in a few pages.) To find out what's happening on the keyboard while our program is running, we must learn to talk with the 8042 processor — or, rather, how to make it talk to us.

When a key is pressed on the keyboard, the 8042 sends a signal called an interrupt to the computer's CPU. An interrupt is a signal that causes the computer to quit what it's doing and execute a special subroutine called an interrupt handler. There are a number of different interrupts that can occur in the course of a PC's daily operations. This particular interrupt is known as the keyboard interrupt. (Officially, the keyboard interrupt is designated as interrupt 09H.)

There is a standard keyboard interrupt handler routine built into the ROM BIOS that came with your computer. Ordinarily, it is this routine that is executed when a key is pressed on your keyboard. When executed, this routine sends a message to the 8042 processor requesting an identifying number, called a scan code, for the last key pressed. The scan code identifies the physical key that has been pressed. There are 101 keys on most PC keyboards and the 8042 uses a unique scan code to identify each one of them. It's important to realize that the 8042 can identify keys *only* by scan code. It has no idea what symbols, such as letters of the alphabet or punctuation marks, the keys represent. When the BIOS keyboard interrupt handler receives a scan code from the 8042, it consults a symbol table to determine what symbol is represented by that key. (This table can easily be altered, incidentally, allowing users to reprogram the keys on the keyboard to represent any symbols in the standard IBM character set. Thus a user

who is tired of the QWERTY keyboard and wants to experiment with a Dvorak keyboard can easily do so by running a simple program, though the letters emblazoned on the keycaps will remain unchanged.)

After receiving the scan code from the 8042, the BIOS keyboard interrupt handler stores it in an array called the keyboard buffer. Then, when an application program calls the INT 16h functions in the ROM BIOS to find out which key was last pressed, the BIOS removes the scan code from the buffer and returns it to the program, along with the ASCII code for the symbol that is represented by that key. For example, if the last key pressed was (A), the keyboard routine will return the scan code 30, which is the scan code for the key that ordinarily has the letter "A" on top of it, along with the ASCII code 97, which is the code for the lowercase letter "a." (If the (SHIFT) key was pressed simultaneously with the "A" key, the ASCII code for an uppercase letter "A" will be returned instead. Although I said earlier that the BIOS routines could not tell us about the simultaneous pressing of two keys, the (SHIFT) key is an exception to this rule.)

The keyboard buffer can hold 15 scan codes. Should it fill up before an application program asks the BIOS to remove these codes, the keyboard interrupt handler will refuse to place additional scan codes in the buffer and will cause the PC speaker to emit an obnoxious beeping noise every time the user presses a key. If you've used a PC for more than ten minutes, you've probably heard this noise several dozen times. A side benefit of writing your own keyboard handler is that the users of the games you write will never, ever have to listen to that noise while playing them!

Swiping the Interrupt

Before we can write our own keyboard handler, we'll need a way to steal the keyboard interrupt away from the BIOS routine that's already handling it. Fortunately, this is simple to do. There's a routine in the ROM BIOS that allows us to install our own interrupt handlers, replacing if necessary the ones that are already present in the BIOS. We can call this routine to install our custom keyboard handler during the initialization phase of our game, then call it again when the program ends to put the old keyboard handler back in place. (The users of the game would be less than thrilled if they returned to DOS only to discover that our keyboard handler was still running — especially when they tried to type a command at the C> prompt and nothing appeared on the display.)

Our collection of low-level keyboard handler routines will consist of three assembly language functions: one to install the new keyboard handler, one to remove it, and the keyboard handler itself. We'll call these functions *initkey()*,

assembly language. (Technically, *newkey()* isn't actually a function, since we'll never call it directly from C/C++. It will be called automatically when a keyboard interrupt is generated — that is, when the user presses a key.) We'll place these three functions in an assembly language module called IO.ASM, along with the other low-level input functions that we'll develop in this chapter.

The initkey() Function

The BIOS routine that installs a new interrupt handler is one of a large set of routines in the PC ROM known as the INT 21H routines, after the assembly language instruction that is used to call them. Specifically, it is function 25h, otherwise known as Set Interrupt Vector. Before we can call this function, however, we must call function 35h, also known as Get Interrupt Vector, to learn the number of the old keyboard handler routine — so that we can restore it when the program terminates. And to store the address of the old function, we'll need to set aside 16 bits of memory for the segment and 16 bits for the offset.

The last of those tasks is the easiest. In assembly language, we can allocate 16-bit data storage with the DW (Define Word) directive, like this:

```
intofs   dw 0
intseg   dw 0
```

These two DW directives allocate a pair of 16-bit storage locations (equivalent to *int* variables in C/C++) called *intofs* and *intseg*. We'll use the first to store the offset of the old interrupt vector and the second to hold the segment. We'll also need a memory location for storing information about which keys are being pressed. This location will need to be accessible to both our C/C++ routines and the keyboard handler, so we'll define it in C++ as an array and pass its address to the *initkey()* function, which will store this address where it can be accessed by the *newkey()* handler. We'll define storage for the address of this array the same way we assigned storage for the address of the old interrupt handler:

```
bufseg   dw 0
bufofs   dw 0
```

The first of these will hold the segment of the array, the second the offset. The actual form that this array will take will be explained in a moment, when we discuss the keyboard handler function.

Next, we'll call INT 21h function 35h using the same procedure we demonstrated for calling DOS and BIOS functions in the last chapter:

```
mov   ah,35h          ; Call INT 21h Function 35h
mov   al,09h          ; ...to get current address of
int   21h             ; ...interrupt 09h
```

The 35h placed in the AH register is the number of the function that will give us the address of the old interrupt handler; the 09h in AL is the number of the function that the handler handles. On return from this call to DOS, the BX and ES registers will contain the offset and segment of the old interrupt handler, respectively. We can move these values directly into the storage locations we've set aside for them:

```
mov   intseg,es          ; Save segment
mov   intofs,bx          ; ...and offset
```

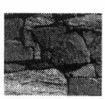
To install a new interrupt handler, we must place the segment and address of the new handler in the DS and DX registers, then call INT 21h function 25h:

```
mov   dx,seg _newkey     ; Put the segment of new
mov   ds,dx              ; ...interrupt handler in DS
mov   dx,offset _newkey  ; Plus offset in DX
mov   ah,25h             ; Call INT 21h Function 25h
mov   al,09h             ; ...to install new handler
int   21h
```

Finally, we'll take the address of the buffer in which key data will be passed back to our C/C++ program and store it in the locations we set aside for it earlier. The address will have been passed to the *initkey()* function in a 32-bit parameter called *buffer*, so we'll move the segment portion and address portion of that parameter into storage:

```
les   dx,buffer          ; Save address of new scan code buffer
mov   bufseg,es
mov   bufofs,dx
```

And then we'll return to C++. The complete *initkey()* function is in Listing 3-1.



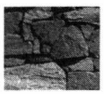
Listing 3-1 The initkey() function

```
_initkey  PROC
; Initialize keyboard interrupt handler
  ARG   buffer:DWORD     ; Pointer to scan code buffer
  push  bp               ; Save BP register
  mov   bp,sp
  push  es               ; Save ES and DS registers
  push  ds
  mov   ah,35h           ; Call INT 21h Function 35h
  mov   al,09h           ; ...to get current address of
  int   21h              ; ...interrupt 09h
  mov   intseg,es        ; Save segment
  mov   intofs,bx        ; ...and offset
  mov   dx,seg _newkey   ; Put the segment of new
```

```
  mov   ds,dx             ; ...interrupt handler in DS
  mov   dx,offset _newkey ; Plus offset in DX
  mov   ah,25h            ; Call INT 21h Function 25h
  mov   al,09h            ; ...to install new handler
  int   21h
  les   dx,buffer         ; Save address of new scan code buffer
  mov   bufseg,es
  mov   bufofs,dx
  pop   ds                ; Restore registers
  pop   es
  pop   bp
  ret
_initkey  ENDP
```

The remkey() Function

The *remkey()* function, which removes the new keyboard handler and reinstalls the old one, is extremely simple. It loads the offset and segment of the old handler into DX and DS and calls INT 21h function 25h again. Aside from the usual set-up and return instructions, that's all it does. The complete function appears in Listing 3-2.



Listing 3-2 The remkey() function

```
_remkey PROC
  push  ds              ; Save DS register
  mov   dx,intseg       ; Get segment and offset of
  mov   ds,dx           ; ...old interrupt handler
  mov   dx,intofs
  mov   ah,25h          ; Call INT 21h Function 25h
  mov   al,09h          ; ...to restore old handler
  int   21h
  pop   ds              ; Restore DS register
  ret
_remkey ENDP
```

The newkey() Function

The most important of the three low-level keyboard functions is the *newkey()* function. This is the actual interrupt handler that we installed and removed with the last two functions. It's written pretty much like any other assembly language procedure, except that (1) we can't pass any parameters to it (because we'll never actually call it), (2) we must end it with an IRET (Interrupt RETurn) instruction, and (3) we must be careful to save the value of every CPU register that we use in

the course of the procedure. The reason for this last requirement is that an interrupt handler such as this will be executed at unexpected times, while the computer is performing other tasks. We must be careful not to mess up the values stored in the registers lest we interfere with whatever task is in progress when the user presses a key. We must leave all registers *precisely* the way we find them.

Once we've finished saving the registers, however, we can get down to the serious business of handling the interrupt. First, we need to get the address of the buffer that we passed from C++ earlier and put it into a register (or, rather, pair of registers). This buffer is a 128-byte array of type *char* in which each position corresponds to one of the 101 scan codes that can be returned from the PC 101-key keyboard. (See Figure 3-1 for a chart showing which scan codes represent which keys.) Of course, since the keyboard has 101 keys and the array has 128 elements, that leaves a few unused spaces in the buffer. But we'll leave those extra 27 entries in place for future keyboard expansion (and so that we won't accidentally trash memory beyond the end of the array should the keyboard processor accidentally send us nonsense data). The design of the PC keyboard is such that it can never have more than 128 scan codes, so we should be safe into the foreseeable future. We'll put the segment and offset of the buffer into registers ES and DI, where we can use them as a pointer to the start of the array:

```
mov   ax,bufseg       ; Point ES:DI at scan code buffer
mov   es,ax
mov   di,bufofs
```

Next, we'll get the scan code for the last key pressed from the keyboard processor. This is done by inputting data through data port 60h. (See Appendix B on machine language programming for more detail on inputting data through ports.) We'll input the data to register AL:

```
in    al,60h          ; Get the latest scan code
```

In a moment we're going to use the 16-bit value in the AX register as an index into the scan code buffer, based on the value that we just input to the AL register. Since AL is already part of the AX register — see Appendix B for details — the job of setting up the index is half done, but we'll need to put a 0 in the other half of the register, like this:

```
mov   ah,0            ; Zero high byte of AX
```

The scan code is now in the AL register. This code will be either a make code or a break code — that is, it will either indicate that the key corresponding to the code has just been pressed or that it has just been released. How can we tell the difference? A break code is equal to the scan code for the key plus 128, while a make code is equal simply to the scan code. Since no scan code on the PC keyboard is allowed to have a value greater than 127 (and at present none have

Key	Code	Key	Code	Key	Code
Esc	1	A	30	Caps Lock	58
! or 1	2	S	31	F1	59
@ or 2	3	D	32	F2	60
# or 3	4	F	33	F3	61
$ or 4	5	G	34	F4	62
% or 5	6	H	35	F5	63
^ or 6	7	J	36	F6	64
& or 7	8	K	37	F7	65
* or 8	9	L	38	F8	66
(or 9	10	: or ;	39	F9	67
) or 10	11	" or '	40	F10	68
_ or -	12	~ or `	41	F11	133
+ or =	13	Left Shift	42	F12	134
Bksp	14	| or \	43	NumLock	69
Tab	15	Z	44	Scroll Lock	70
Q	16	X	45	Home or 7	71
W	17	C	46	Up or 8	72
E	18	V	47	PgUp or 9	73
R	19	B	48	Gray -	74
T	20	N	49	Left or 4	75
Y	21	M	50	Center or 5	76
U	22	< or ,	51	Right or 6	77
I	23	> or .	52	Gray +	78
O	24	? or /	53	End or 1	79
P	25	Right Shift	54	Down or 2	80
{ or [	26	Prt Sc or *	55	PgDn or 3	81
} or]	27	Alt	56	Ins or 0	82
Enter	28	Spacebar	57	Del or .	83
Ctrl	29				

Figure 3-1 The scan codes for the 101-key keyboard

values greater than 108), this means that make codes have values in the range 0 to 127, and break codes have values in the range 128 to 255.

The next thing that our keyboard handler must do, then, is to check to see which kind of code it has received. We'll use the CMP (compare) instruction to compare the value in AL with 128:

```
cmp   al,128          ; Was code a "make" or a "break"?
```

If it's a break code (that is, if a key has been released), this instruction will set the CPU borrow flag (see Appendix B). We can make a jump instruction

contingent on this flag and branch to the routine for handling break codes if the flag is set:

```
jnb   break                   ; If break, skip ahead
```

If this jump isn't taken, then it's a make code — that is, a key has been pressed. We'll move the value in AX to BP, where we can use it as an index into the buffer. And we'll store a 1 in the buffer element corresponding to this scan code, to indicate that the key is currently pressed:

```
mov   bp,ax                   ; Else use code as index into buffer
mov   [byte ptr es:di+bp],1   ; And flag key as pressed
jmp   endkey
```

That last instruction skips over the instructions that handle break codes. These instructions are just like those that handle the make code, except that they first subtract 128 from the scan code (by ANDing the value with 127; see Appendix B) and then move a 0 into the appropriate buffer position rather than a 1:

```
break:
  and   al,127                 ; If break code, remove high bit
  mov   bp,ax                  ; Use code as index into buffer
  mov   [byte ptr es:di+bp],0 ; Flag key as released
```

The next four instructions, which are performed whether or not the code was a make or a break, change the value of a single binary digit of the value received through input port 61h, then send this value back out through that port. This rather odd operation "clears" the interrupt request — that is, it tells the 8042 processor in the keyboard that we won't be needing further copies of the most recently received scan code. (It's possible to write keyboard handler routines that "chain" to the regular BIOS keyboard handler by not clearing the interrupt request and passing control to the regular handler once the custom handler does its thing. This, for instance, is the method that terminate-and-stay-resident utilities — TSRs — use to watch for the pressing of certain key combinations while not disturbing the activity of the regular keyboard handler.)

Here's how the clearing operation is performed:

```
endkey:
  in    al,61h                 ; Clear the keyboard interrupt
  mov   ah,al
  or    al,80h
  out   61h,al
```

We must also send a message to the PC's general interrupt handling circuitry, which up until now has been refraining from sending any additional interrupts to the CPU while we performed our interrupt processing, telling it that the computer is ready to process other interrupts. This is done by sending a value of 20h through output port 20h:

```
mov  al,20h           ; Allow additional interrupts
out  20h,al
```

The rest of the interrupt handler consists of restoring the value of all saved registers and executing an IRET instruction. The complete text of the keyboard interrupt handler appears in Listing 3-3.

Listing 3-3 The newkey() function

```
_newkey  PROC FAR
  push  ax                    ; Save all registers used
  push  di
  push  bp
  push  es
  mov   ax,bufseg             ; Point ES:DI at scan code buffer
  mov   es,ax
  mov   di,bufofs
  in    al,60h                ; Get the latest scan code
  mov   ah,0                  ; Zero high byte of AX
  cmp   al,128                ; Was code a "make" or a "break"?
  jnb   break                 ; If break, skip ahead
  mov   bp,ax                 ; Else use code as index into buffer
  mov   [byte ptr es:di+bp],1 ; And flag key as pressed
  jmp   endkey
break:
  and   al,127                ; If break code, remove high bit
  mov   bp,ax                 ; Use code as index into buffer
  mov   [byte ptr es:di+bp],0 ; Flag key as released
endkey:
  in    al,61h                ; Clear the keyboard interrupt
  mov   ah,al
  or    al,80h
  out   61h,al
  mov   al,20h                ; Allow additional interrupts
  out   20h,al
  pop   es                    ; Restore registers
  pop   bp
  pop   di
  pop   ax
  iret
_newkey  ENDP
```

Using the Keyboard Functions

These three low-level keyboard functions — *initkey()*, *remkey()*, and *newkey()* — must always be used together. And they must be used with considerable caution. When our custom keyboard handler snaps into place, it will be the *only* means we

have of communicating with our program. Even the familiar (CTRL)-(ALT)-(DEL) combination will cease to work. If you're debugging a program under Turbo Debugger or the built-in debugger that comes with the Borland C++ IDE, you'll no longer be able to terminate the program by hitting (CTRL)-(ESC). So it's imperative that you put some kind of escape clause into your program before the first time you run it, even if you're working within a familiar testing and debugging environment. For instance, you can add code to your program that will watch for the keyboard handler to report that the (ESC) key (scan code 1) has been pressed.

Here's how you can incorporate the keyboard handler into a program. Link the IO.ASM file to your code and include the IO.H header file at the top of any modules that reference the *initkey()* or *remkey()* functions. Put a call to *initkey()* near the beginning of your code, like this:

```
initkey(scanbuffer);
```

where *scanbuffer* is a 128-byte array of type *char* with non-local scope — that is, it has been declared externally to any functions. From that point until you invoke the *remkey()* functions, the scan codes of all keys pressed on the keyboard will be instantly (and rather mysteriously) flagged in the *scanbuffer* array. For instance, if the (ENTER) key (which has a scan code of 28) is pressed, element 28 of this array will be set to 1 for as long as it is held down. It will be reset to 0 when the key is released. Your program can test these elements to see which keys are currently being pressed.

The SCANKEY Program

Now let's test our keyboard handler to see if it works as expected. The short program in Listing 3-4 (available on the disk as SCANKEY.EXE) will repeatedly print the word "ENTER" on the display as long as the (ENTER) key is being held down and the word "SHIFT" as long as the (SHIFT) key is held down. If both keys are pressed simultaneously, it will print both words alternately. When the (ESC) key is pressed, it terminates. It does all of this simply by installing the keyboard handler and watching for elements 1, 28, and 42 (scan codes for (ESC), (ENTER), and (SHIFT)) to be set to 1. (Actually, it watches for them to be set to any non-zero value.) The SCANKEY output looks like this:

```
SHIFT
SHIFT
SHIFT
SHIFT
SHIFT
SHIFT
SHIFT
```

```
SHIFT
SHIFT
SHIFT
ENTER
ENTER
ENTER
ENTER
ENTER
ENTER
ENTER
ENTER
ENTER
ENTER
ENTER
ENTER
ENTER
```

Listing 3-4 The SCANKEY.CPP program

```
// SCANKEY.CPP
//
// Demonstrates alternate keyboard handler
//
// Written by Christopher Lampton
// for Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  "io.h"

char far scanbuffer[128]; // Buffer for 128 scan codes

void main(void)
{
  // Install alternate keyboard driver:
  initkey(scanbuffer);

  // Loop until ESC (scan code 1) is pressed:
  while(!scanbuffer[1]) {

    // Watch for SHIFT key:
    if (scanbuffer[42]) printf("SHIFT\n");

    // Watch for ENTER key:
    if (scanbuffer[28]) printf("ENTER\n");
  }

  // Remove alternate keyboard handler:
  remkey();
}
```

Removing the Keyboard Handler

Once you're finished with the keyboard handler, remove it by calling the *remkey()* function, like this:

```
remkey();
```

Forget to do that, and the computer will appear to lock up when your program ends. Of course, you may not notice this if you're running under Turbo Debugger or the Borland IDE at the time, since both of these programs are aggressive about re-installing their own keyboard handlers when a program terminates. But most users of your program will be running it from the DOS prompt, where applications are expected to observe an honor system about reinstalling the old keyboard handler when their work is done. Later in this chapter, we'll show you how to incorporate these low-level keyboard handling routines into a higher-level interface, in this case an event management module for a wireframe maze program. For the moment, however, we're going to turn to another input device, the PC mouse.

The PC Mouse

For microcomputer users, the input device known as the mouse has always been more closely identified with the Apple Macintosh computer than with the IBM PC and its clones. That's because the mouse has been standard equipment with all Macintoshes since the first Mac rolled off the assembly line more than a decade ago. It's pretty much impossible to use a Macintosh without one.

PC users, by contrast, have come to the mouse in dribs and drabs. Until recently, most PC software used the keyboard as the primary input device, with mouse support added as an afterthought (if at all). But the increasing popularity of the Microsoft Windows operating system has changed that. Like the Macintosh operating system, Windows is expressly designed to be used with a mouse. And, as Windows has found its way onto more and more computer systems, so have mice. Now it's a rare PC-compatible computer that doesn't have one attached to it.

Game designers have taken note of this phenomenon. More and more, the mouse is being made available as an optional control device on PC-based games. Many PC games are now being designed with the mouse as a primary input device, though keyboard control is still made available for those users without a rodent attached to their system unit. Dungeon Master, the seminal maze game mentioned in chapter 1, is quite difficult to control without a mouse. This is due in part to its heritage on the Atari ST, a computer that (like the Macintosh) comes with a mouse as standard equipment, but games inspired by Dungeon

Master — SSI's Eye of the Beholder series, for instance — also tend to have a strong mouse orientation. Although a maze game can be designed without mouse support, it is increasingly essential that the maze game programmer have a background in mouse-programming skills.

The Mouse Driver

You'll be happy to know that we don't have to design our own low-level handler for the mouse the way we did for the keyboard. That's because a low-level mouse handler is supplied with every mouse sold. It's called the mouse driver and is generally loaded into the computer's memory when the CONFIG.SYS and AUTOEXEC.BAT files are executed at power-up time. (Some mouse drivers are booted from CONFIG.SYS and others from AUTOEXEC.BAT, depending on how the driver itself has been implemented.) To receive information from the mouse, we have only to call this driver and tell it what we want to know. There's a standard interface that programs can use to talk to the mouse driver, so this is actually a very simple process. (Some early mouse drivers, which did not follow the standard for such drivers set by the Microsoft Mouse, may not respond to the standardized set of mouse driver commands, but these are now fairly rare and can generally be ignored. However, you should specify that your programs require a Microsoft-compatible mouse lest some user with a prehistoric driver complain that your program won't work on his or her machine.)

Like the input routines in the BIOS, the mouse driver is best accessed from assembly language, since interfacing to the mouse driver from C++ is awkward at best. So we'll add a few short mouse routines to our IO.ASM file, which can then be called as functions from a C/C++ program. Although these routines will barely scratch the surface of the tricks that the mouse can perform — they'll only call three of the many functions available from most mouse drivers — they will nonetheless provide us with enough information to allow us to use the mouse in our programs.

The mouse driver functions that we will be calling are listed in Table 3-1. These functions are designed to be called from an assembly language program using the INT 33H instruction. The number of the desired function must be placed in the AX register before the call. Additional information is exchanged in the other registers listed in the table. The first of these functions, function 0, simply initializes the mouse driver. (It also tells us how many buttons the mouse has, since some PC mice have two buttons and others have three.) Function 3 tells us whether or not any of the three mouse buttons is being pressed. (It also returns a pair of numbers representing the current position of the mouse, but we're going to ignore these in favor of the information returned by the next function.) Function 0Bh tells us how far the mouse has moved across the user's

desktop since the last time the function was called and in what directions it has moved. This relative position is measured in a unit called, for obvious reasons, the *mickey*, which is equivalent to 1/400th of an inch. (For older mice, the mickey is equal to 1/200th of an inch.)

To access these functions, we'll create three assembly language mouse functions: *initmouse()* to initialize the mouse interface, *readmbutton()* to detect whether the mouse button has been pressed, and *relpos()* to read the position of the mouse relative to the last position reported.

Table 3-1

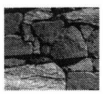
```
INT 33H
 Function 00H
  Reset mouse driver
   IN:
    AX: 0000H
   OUT:
    AX: Initialization status
      FFFFH: Successfully installed
      0000H: Installation failed
    BX: Number of mouse buttons

 Function 03H
  Get pointer position and button status
   IN:
    AX: 0003H
   OUT:
    BX: Button status
     Bit 0: Left mouse button (1 if pressed)
     Bit 1: Right mouse button (1 if pressed)
     Bit 2: Center mouse button (1 if pressed)
    CX: X coordinate
    DX: Y coordinate

 Function 0BH
  Get relative mouse position
   IN:
    AX: 000BH
   OUT:
    CX: Relative horizontal distance (in mickeys)
    DX: Relative vertical distance (in mickeys)
```

Table 3-1 The mouse driver functions called by the routines in this chapter

The initmouse() Function

The *initmouse()* function mostly just loads a 0 into the AX register and calls the mouse driver. While it may not be obvious from looking at the function itself, it also returns an error code as the integer value of the function, since the mouse driver leaves this code in the AX register and the contents of this register are automatically passed back to C/C++ as the result of the function call. A 0 value means that an error has occurred.

Listing 3-5 contains the text of the *initmouse()* function.



Listing 3-5 The initmouse() function

```
_initmouse      PROC
; Call mouse driver initialization routine
  mov  ax,0             ; Request function zero (initialize)
  int  33h              ; Call mouse driver
  ret                   ; Return with error code in AX register
_initmouse      ENDP
```

The readmbutton() Function

The function that returns information about the mouse buttons isn't much more complicated. The *readmbutton()* function calls mouse driver function 3, which reports on the absolute position of the mouse and the status of the buttons. We're not interested in the absolute position of the mouse, so this value is ignored. A single byte value describing the current status of the mouse buttons is returned from function 3 in the BX registers, so we move that value into the AX register where it is returned to C/C++ as the integer value of this function. The first three individual bits — binary digits — in this 16-bit binary number tell us whether the equivalent mouse buttons are pressed or not. If a button is pressed, the equivalent bit is set to 1. If not, the bit is reset to 0. The first (or rightmost) bit of the number represents the status of the left mouse button, the second bit represents the status of the right mouse button, and the third bit represents the status of the center mouse button. We'll show you more about how to read these bits in a moment. The text of the *readmbutton()* function appears in Listing 3-6.

Listing 3-6 The readmbutton() function

```
_readmbutton  PROC
; Read mouse button
```

continued on next page

continued from previous page

```
  mov     ax,3            ; Request function 3 (read buttons)
  int     33h             ; Call mouse driver
  mov     ax,bx           ; Put result in function return
                          ;  register
  ret
_readmbutton  ENDP
```

The third button isn't present on all PC mice, so you shouldn't assume that the user has one. Some games assign an inessential function to this button, often one that can also be accessed from the keyboard, but the majority of games ignore it completely.

The relpos() Function

Finally, we want to know how far the mouse has moved since the last time we checked up on its whereabouts. The *relpos()* function obtains that information by calling mouse driver function 0bh. The mouse can move in two directions, roughly corresponding to the x and y axes of a Cartesian coordinate system, so the mouse driver will return to us two values representing the relative position of the mouse in each direction, as measured in *mickeys*. Negative values represent movement to the left or up while positive values represent movements to the right or down. (Note that this corresponds to the way in which Cartesian coordinates on the video display traditionally grow smaller to the left and up and larger to the right and down. See Figure 3-2.)

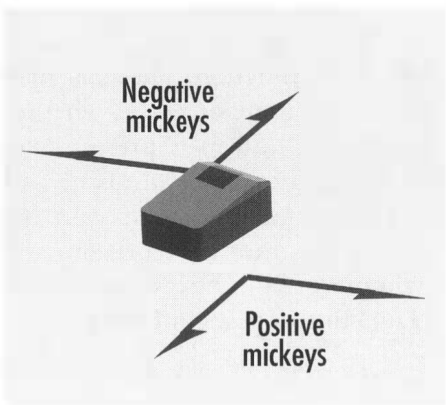


Figure 3-2 Negative mickeys represent movement up and left; positive mickeys represent movement down and right

Since the *relpos()* function will need to return both of these values to the C/C++ routine that is calling it, we'll need to pass the function pointers to a pair of integer variables in which it can return these values. We'll do this using the *ARG* directive, as described in the last chapter (and in the appendix on assembly language). After calling the mouse driver, the *relpos()* function will retrieve the two values from the CX and DX registers, place them in the variables pointed to by these pointers, and return to the caller. The text of the *relpos()* function appears in Listing 3-7.

Listing 3-7 The relpos() function

```
_relpos   PROC
; Get changes in mouse position relative to last call
  ARG   x:DWORD,y:DWORD
  push  bp
  mov   bp,sp
  mov   ax,000bh      ; Request function 0bh
                      ;  (relative mouse position)
  int   33h           ; Call mouse driver
  les   bx,x          ; Point es:bx at x parameter
  mov   [es:bx],cx    ; ...and store relative position
  les   bx,y          ; Point es:bx at y parameter
  mov   [es:bx],dx    ; ...and store relative position
  pop   bp
  ret
_relpos   ENDP
```

Why would we be interested in the *relative* position of the mouse, as opposed to its *absolute* position? In this book, we won't be using the mouse to guide a pointer on the display, the way that some programs do. Instead, we'll be using it much the way we use the joystick, to indicate directions of motion. Thus all we'll really need to know is the direction in which the mouse is moving, which can easily be determined from its relative position. More about this in a moment.

Using the Mouse Functions

To use these three mouse functions in a program, link IO.ASM to your code and include the IO.H file in all modules that reference them. Initialize the mouse during the initialization of the program by calling *initmouse()*, like this:

```
if (!initmouse()) exit(1);
```

You'll notice that this instruction exits the program with an error code if the *initmouse()* function returns a 0 value. (You'll recall that this means an error

occurred in initializing the mouse.) Alternatively, if the mouse isn't crucial to your game, you could set a flag indicating that the mouse isn't available for use, then continue with the program, like this:

```
if (initmouse()) mouse_input=FALSE;
```

To find out if the left or right mouse buttons are currently being pressed, call the *readmbutton()* function, like this:

```
int button_status = readmbutton()
```

If the left mouse button is being pressed, the first bit — that is, the rightmost binary digit — of the integer variable *button_status* will be set to 1. You can test for this by performing a bitwise AND between *button_status* and the constant number 1, then testing the result to see if it's non-zero, like this:

```
if (button_status & 1)
  printf("The left mouse button is being pressed.\n");
```

If the right mouse button is being pressed, the second bit of *button_status* will be set to 1. You can similarly test for this by performing a bitwise AND between *button_status* and the constant number 2. The predefined constants LMBUTTON and RMBUTTON, equal to 1 and 2 respectively, are provided in the IO.H file for this purpose.

Finally, to determine if the mouse has moved since you last checked, you can call the *relpos()* function, passing pointers to a pair of integer variables to it as parameters, like this:

```
int xrel,yrel;
relpos(&xrel,&yrel);
```

On return from this function, *xrel* will contain the number of mickeys that the mouse has moved in the x direction since the last call to this function, with negative values representing movements to the left and positive values representing movements to the right. Similarly, *yrel* will contain the number of mickeys that the mouse has moved in the y direction since the last call to this function, with negative values representing upward movements and positive values representing downward movements.

The MOUSE Program

To demonstrate the mouse functions, Listing 3-8 contains a short program called MOUSE.CPP, available on the disk as MOUSE.EXE, which prints the x and y coordinates of the mouse and the status of the mouse buttons on the text display, updating them continuously as you move the mouse and press the buttons. (Pressing the right mouse button also terminates the program, so you'll only see the pressing of this button reported briefly before the program ends.) It does this

by assuming the initial position of the mouse to be at coordinates 0,0, then calling *relpos()* on every pass through a loop and adding the relative changes in the mouse position to these coordinates. At the same time, it calls *readmbutton()* to check for the pressing of the mouse button. The *gotoxy()* library function, prototyped in the CONIO.H header file, is used to format this information neatly in the middle of the text screen. (The *clrscr()* function, prototyped in the same file, is used to clear the display while the program is initialized.) The MOUSE output looks like this:

```
Mouse X    442
Mouse Y= -279
Left button not pressed
Right button not pressed
```

Listing 3-8 The MOUSE.CPP program

```
// MOUSE.CPP
//
// Demonstrates mouse interface functions
//
// Written by Christopher Lampton
// for Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <stdlib.h>
#include  <conio.h>
#include  "io.h"

void main(void)
{
  int bstatus,xrel,yrel;
  int mousex=0,mousey=0; // Mouse x,y positions

  // Clear text display:
  clrscr();

  // Initialize mouse driver:
  if (!initmouse()) exit(1);

  // Read mouse button status:
  bstatus=readmbutton();

  // Loop until right mouse button is pressed:
  while(!(bstatus & RMBUTTON)) {

    // Read relative mouse position:
```

continued on next page

continued from previous page

```
    relpos(&xrel,&yrel);

    // Update mouse position variables:
    mousex+=xrel;
    mousey+=yrel;

    // Write mouse positions on display:
    gotoxy(20,10);
    printf("Mouse X=%6d",mousex);
    gotoxy(20,11);
    printf("Mouse Y=%6d",mousey);

    // Read mouse button status:
    bstatus=readmbutton();

    // Write button status on display
    gotoxy(20,12);
    if (bstatus & LMBUTTON) printf("Left button pressed    ");
    else printf("Left button not pressed");
    gotoxy(20,13);
    if (bstatus & RMBUTTON) printf("Right button pressed    ");
    else printf("Right button not pressed");
  }
}
```

The PC Joystick

Finally, we come to the joystick. It's hard to think of the joystick as being a strange and obscure input device — joysticks for home gameplay have been around at least since the Atari 2600 game machine was released back in the 1970s — yet it's probably the hardest input device to find solid information about. This may be because it's only used for game input and not for "serious" purposes (though flight simulator fans, who have been known to take their hobby *very* seriously, may disagree with this summation). High-minded programming texts usually don't contain information on such frivolous devices as the joystick.

There are a couple of joystick functions in the PC's BIOS, but they aren't especially useful. As with the keyboard, we'll write our own assembly language joystick routines that interact directly with the joystick hardware. We'll put these functions into IO.ASM, along with our growing array of assembly language I/O functions for other PC input devices.

Analog Joysticks

The PC joystick is an analog joystick. This means that it can give us information that the joysticks attached to certain other computers, such as the Atari ST and

Commodore Amiga, cannot. Not only can it tell us whether the user is pressing the joystick buttons and pushing the stick up, down, right, or left — but it can tell us how *far* the user is pushing the joystick. Alas, this makes the joystick rather difficult to program.

On an assembly language level, we can receive information about the joystick by inputting data through port 0201h, with a series of instructions such as this:

```
mov  dx,0201h  ; Put gameport address in DX
out  dx,al     ; Send random data through port
in   al,dx     ; Get valid data from port
```

The first instruction establishes 0201h as the number of the port we wish to send data through. The second instruction outputs random data through the port, which signals to the joystick circuitry that we wish to receive information about the stick position. And the third instruction receives data from the joystick and places it in register AL.

The value that we receive from the joystick with these instructions is known as the gameport byte. The individual bits in the gameport byte give us information about the position of the joystick — in fact, they can give us simultaneous information about *two* joysticks — but they do not yield this information easily. They also give us information about whether the joystick buttons are being pressed. This information is more easily decoded, so we'll talk about it first.

Reading the Joystick Buttons

Two joysticks can be attached to the PC gameport at one time; we'll call these joystick A and joystick B. Each joystick has two buttons, which we'll call button 1 and button 2. This means that we can read the status of up to four different buttons in the gameport byte.

Bits 4 and 5 of the gameport byte represent the status of buttons 1 and 2 on joystick A. Bits 6 and 7 of the gameport byte represent the status of buttons 1 and 2 on joystick B. When one of these bits is set to 0, the equivalent button is being pressed. When one of these bits is set to 1, the equivalent button is *not* being pressed.

This is not exactly intuitive. One would expect it to work the other way around, with a 1 bit representing a button that's being pressed and a 0 bit representing a button that *isn't* being pressed. So we'll flip the bits in the gameport byte before we send this information back to the calling program. We can do this with the assembly language NOT instruction, which changes 0s to 1s and 1s to 0s. A short assembly language routine for reading the gameport byte, flipping the appropriate bits, and passing the resulting value back to a calling C/C++ program is shown in Listing 3-9. The calling program must provide a parameter for this function, telling it which joystick buttons the function should read. This parameter consists of a binary "mask" in which those bits are set to 1

that correspond to the bits in the gameport byte that the program is interested in reading. We'll show you how to use this function in a moment. Note that the gameport is referred to in this function by the constant GAMEPORT, which is defined elsewhere in the IO.ASM module as equal to 0201h.

Listing 3-9 The readjbutton() function

```
_readjbutton  PROC
; Read joystick button specified by BMASK
  ARG  bmask:WORD
  push bp
  mov  bp,sp
  mov  dx,GAMEPORT     ; Point DX at joystick port
  mov  ah,0            ; Zero high byte of return value
  out  dx,al           ; Request data from port
  in   al,dx           ; Get value from joystick
  not  al              ; Flip button bits
  mov  bx,bmask        ; Mask out all but requested buttons
  and  al,bl           ; And leave result in AX
  pop  bp
  ret
_readjbutton  ENDP
```

Determining the Stick Position

Reading the position of the stick itself is a great deal more difficult. The position of joystick A on the x axis — that is, in the left and right direction — is represented by bit 0 of the gameport byte while the position of joystick A on the y axis — that is, in the up and down direction — is represented by bit 1 of the gameport byte. Similarly, the position of joystick B on the x axis is represented by bit 2 of the gameport byte, while the position of joystick B on the y axis is represented by bit 3 of the gameport byte.

How can a single bit in the gameport byte represent the position of a joystick? With great difficulty. When we first write random data to port 0201h and read the value of the gameport byte, the bits representing the various stick axes will *always* be set to 1. So we must continue reading the gameport byte from port 0201h in a loop, until the bit corresponding to the stick axis that we are interested in is reset to 0. The amount of time that it takes for this to happen will tell us what position the stick is in on that axis. If it resets quickly, the stick is pulled to the left or up (depending on which axis we are measuring). If it resets slowly, the stick is pulled to the right or down. If it resets in an intermediate amount of time, the stick is somewhere in the middle. Thus we must time the

number of loops required for the appropriate bit to reset in order to determine the stick position.

The easiest way to do this is with the aid of two 80x86 assembly language instructions: TEST and LOOPNE. The TEST instruction tests the value of a specific bit in a specific register, while the LOOPNE instruction loops back to an earlier instruction as long as the result of the last operation (which in this case will be the TEST operation) wasn't 0. LOOPNE also subtracts 1 from the value in the CX register every time it loops (and stops looping if that value reaches 0).

If we test the bit of the gameport byte that we are interested in with TEST, we can use the LOOPNE instruction to keep looping until the bit resets to 0. And if we begin looping with a value of 0 in the CX register (which will roll over to FFFFh the first time LOOPNE executes, preventing the loop from terminating immediately), we can prevent the loop from taking forever to terminate — in case something is seriously wrong with the joystick hardware — while keeping a count of the number of times the loop has executed. Of course, since the value in CX is decremented rather than incremented every time the loop is executed, we'll need to subtract the value in CX from 0 to determine the actual number of times the loop repeated before the gameport byte was reset to 0. An assembly language function that does all of these things is in Listing 3-10.

Listing 3-10 The readstick() function

```
_readstick  PROC
; Read current position of joystick on axis specified by BMASK
  ARG  bmask:WORD
  push bp
  mov  bp,sp
  cli                           ; Turn off interrupts, which could
                                ;  effect timing
  mov  ah,byte ptr bmask        ; Get bitmask into ah.
  mov  al,0
  mov  dx,GAMEPORT              ; Point DX at joystick port
  mov  cx,0                     ; Prepare to loop 65,536 times
  out  dx,al                    ; Set joystick bits to 1
loop2:
  in   al,dx                    ; Read joystick bits
  test al,ah                    ; Is requested bit (in bitmask)
                                ;  still 1?
  loopne loop2                  ; If so (and maximum count isn't
                                ;  done), try again
  sti                           ; Count is finished, so reenable
                                ;  interrupts
  mov  ax,0
```

continued on next page

continued from previous page

```
  sub  ax,cx                    ; Subtract CX from zero, to get count
  pop  bp
  ret
_readstick  ENDP
```

Using the Joystick Functions

The joystick functions are both simpler and more complicated to use than the earlier input functions in this chapter. No initialization is required before they can be used. And only two types of data are returned by these functions: stick position data and button data.

Both the *readjbutton()* and *readstick()* functions are called with a single parameter — a bitmask that tells the functions which bits to read in the gameport. So that you don't have to worry about which binary digits are which, we've provided appropriate masks as constants in the IO.H file. These constant masks are JOY_X and JOY_Y, for reading the x and y axes of joystick A, and JBUTTON1 and JBUTTON2 for reading buttons 1 and 2 of joystick A. (You'll have to come up with your own masks for reading joystick B.)

For instance, to read the x axis position of joystick A, call the *readstick()* function like this:

```
int stick1=readstick(JOY_X);
```

This will set the integer variable *stick1* to the number of times the assembly language loop executed when timing the bit 0 of the gameport, which gives you a method of calculating the position of joystick A. (We'll show you how to use this value in a moment.) Similarly, to read the y axis position of joystick A, call the *readstick()* function like this:

```
int stick1=readstick(JOY_Y);
```

To read the status of button 1 on joystick A, call the *readjbutton()* function like this:

```
int button1=readjbutton(JBUTTON1);
```

If the value returned by the function is non-zero, then button 1 is being pressed. If the value is 0, then it isn't being pressed. Similarly, to read the status of button 2 on joystick A, call the *readjbutton()* function like this:

```
int button2=readjbutton(JBUTTON2);
```

If you need to know whether one or the other of the buttons is being pressed, but don't care *which* one, you can read them both at the same time like this:

```
int button=readjbutton(JBUTTON1+JBUTTON2);
```

If the value returned by the function is non-zero, then one or both of the buttons is being pressed. If the value is 0, then neither is being pressed. Note that you can't use this trick with the *readstick()* function to read the value of the stick on both axes simultaneously. Well, you *can*, but the value returned would be meaningless. (Actually, it would be the greater of the x and y axis values, but there would be no way for you to tell which one it was.)

Calibrating the Joystick

The hardest part of using the joystick functions is figuring out what the value returned by the *readstick()* function *means.* Recall that this value is the number of times that the assembly language loop that read the stick value had to loop before the appropriate bit in the gameport reset to 0. But what does this tell you about the physical position of the stick?

That's difficult to say, because on different computers it will mean different things. The faster the computer, the more times the timing loop will execute before the bit resets to 0. So a number that means the joystick is being pushed to the right on a slow machine may mean the joystick is being pushed to the left on a fast machine. How do you determine what these numbers mean for the machine the program is actually running on?

The answer is that you must calibrate the joystick before the game can begin. You must prompt the user to move the joystick to the extreme upper left and perform some action, such as pushing a joystick button, that will signal when the stick is in position. When this signal is received, you must call the *readstick()* function to find out what number is returned for that position. You must then repeat this operation for the center and lower right positions, noting the numbers for each. Once you have these numbers, you can use them for determining the joystick positions while the game is actually being played.

The JOYSTICK Program

Listing 3-11 demonstrates how to use the joystick functions in a program. The logic of the JOYSTICK.CPP program is similar to that of the MOUSE.CPP program earlier in this chapter, though the need to calibrate the joystick makes this listing a bit more complicated. To keep things as simple as possible, we've calibrated only one set of reference values for the joystick — the ones for the center position. The program then loops repeatedly, measuring the joystick position on both axes and announcing if it's to the left or right of, or exactly in, this central position. It also notes whether the buttons are pressed. When the second button is pressed, the program terminates. While running this program

(which is available on the disk as JOYSTICK.EXE), you'll note that it's extremely difficult to get the joystick precisely in the center. In practice, we'll need to represent the center position as a range of values around this reference position, so that slight joystick slippage to either side won't be taken as a command from the user. The output looks like this:

```
X axis position:
  Left of center
Y axis position:
  Above center
Button 1 not pressed
Button 2 not pressed
```

Listing 3-11 The JOYSTICK.CPP program

```
// JOYSTICK.CPP
//
// Demonstrates joystick interface functions
//
// Written by Christopher Lampton
// for Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <stdlib.h>
#include  <conio.h>
#include  "io.h"

void main(void)
{
  int xstick,ystick;
  int xcent,ycent;
  int status1,status2;

  // Clear the text display:
  clrscr();

  // Calibrate the user's joystick:
  printf("\n\nCenter your joystick and press button ");
  printf("one.\n");
  while (!readjbutton(JBUTTON1)); // Loop until joystick
                                  //  button pressed
  xcent=readstick(JOY_X);         // Get x coordinate
  ycent=readstick(JOY_Y);         // Get y coordinate
  while (readjbutton(JBUTTON1));  // Loop until button
                                  //  released
  // Set up display:
  clrscr();
```

```
  gotoxy(20,10);
  printf("X axis position:");
  gotoxy(20,12);
  printf("Y axis position:");

  // Read button 2 status:
  status2=readjbutton(JBUTTON2);

  // Loop until right mouse button is pressed:
  while(!status2) {

    // Read X and Y axis positions:
    xstick=readstick(JOY_X);
    ystick=readstick(JOY_Y);

    // Write joystick positions on display:
    gotoxy(20,11);
    if (xstick<xcent) printf("  Left of center ");
    if (xstick>xcent) printf("  Right of center");
    if (xstick==xcent) printf("  Centered       ");
    gotoxy(20,13);
    if (ystick<ycent) printf("  Above center");
    if (ystick>ycent) printf("  Below center");
    if (ystick==ycent) printf("  Centered    ");

    // Read mouse button status:
    status1=readjbutton(JBUTTON1);
    status2=readjbutton(JBUTTON2);

    // Write button status on display
    gotoxy(20,14);
    if (status1) printf("Button 1 pressed    ");
    else printf("Button 1 not pressed");
    gotoxy(20,15);
    if (status2) printf("Button 2 pressed    ");
    else printf("Button 2 not pressed");
  }
}
```

The Event Manager

The low-level input functions that we have been developing in this chapter can be called directly from a game program to determine what sort of input commands are coming from the user through the input device of choice, be it the keyboard, joystick, or mouse. It would make our lives as programmers a lot easier, though, if there were some sort of protective layer between the game program and the raw data input by these low-level functions, a protective layer that would translate this raw input into a form more appropriate for our game.

That's the job of an event manager. An event manager is a routine or set of routines that monitors the input from the low-level functions and passes on only the information needed by the rest of the program, in a form that the program can easily use. Unlike our low-level input functions, which can be used in pretty much the same form by just about any type of computer program from a spreadsheet to a flight simulator, the event manager will be custom-designed for the type of program with which we intend to use it.

In this chapter we want to write an animated maze exploration program based on the maze-drawing routines presented in the last chapter. It would be appropriate for such a program to receive input from the user concerning the type of movement that the user wishes to make through the maze. For instance, if the user wishes to move forward down a corridor of the maze, he or she could press the ↑ key, push the joystick forward, or move the mouse forward on the desktop. If the user wishes to rotate to the right, he or she could press the → key, push the joystick to the right, or move the mouse to the right. And so forth.

The event manager for such a program could watch for the appropriate actions to be performed, then translate those actions into a set of "movement events" which could be passed back to the game program. If the user pushes the joystick forward, for instance, a "forward movement" event could be returned to the program. Similarly, if the user presses the right arrow key, a "rotate right" event could be returned to the program. The program need not know whether these events were generated by the joystick, the mouse, or the keyboard (or by some even more exotic input device that we've added to the event manager at the last moment, such as a touch screen or data glove). The program only needs to know that the user has requested forward movement, so that it can try to provide what the user wants. The program can then be written without any specific concern for the nature of the input devices that are generating these events.

Maze Events

Before we can write the event manager, then, we need to decide on the set of events that will be required by a simple maze exploration program. The event manager we develop in this chapter will recognize five types of event, which we'll call go_forward, go_back, go_left, go_right, and quit_game. To pass these events from the event manager to the calling program, we'll use an *event structure* that we'll define like this:

```
// Structure for passing events to calling program:
struct event_struct {
  int go_forward,go_back,go_left,go_right,quit_game;
};
```

We'll place this definition in the file EVNTMGR1.H. The "1" in the file name indicates that this will not be the last event manager that we create in this book.

The event structure will be declared by the calling program and passed to the event manager as a parameter. The event manager will then change the value of the fields in the structure to indicate what events have occurred and pass it back to the calling program. When the event manager sets a field in the structure to a non-zero value, it indicates that the event has occurred (or is still occurring, in the case of continuous events such as keys being pressed or sticks being pushed). Because more than one field can be set to a non-zero value at the same time, the event manager can pass multiple events back to the calling program. For example, this structure could be used to indicate that the user wants to move forward and turn right simultaneously.

The init_events() Function

The event manager will be initialized by calling the *init_events()* function, shown in Listing 3-12. This does nothing more than initialize the low-level keyboard and mouse handlers. (The joystick handler doesn't require any initialization, although the user's joystick will need to be calibrated.) The address of a 128-byte array of type *char* called *keybuffer* (defined elsewhere in the event manager module) is passed to the keyboard handler.

Listing 3-12 The init_events() function

```
void init_events()
// Initialize event manager
{
  initkey(keybuffer);     // Install alternate keyboard driver
  initmouse();            // Initialize the mouse driver
}
```

The Joystick Calibration Functions

There are three functions for calibrating the joystick: one that determines the center positions in both axes (the *setcenter()* function), one that determines the farthest up and left positions (the *setmin()* function), and one that determines the farthest down and right positions (the *setmax()* function). The values for these positions are stored in a set of variables that are global to the event manager module, so they can be used later to determine whether the joystick is pointing to

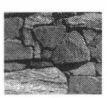
the left or right, up or down. These functions must be called during the initialization of the game, after the user has been instructed to move the joystick into the appropriate positions and press joystick button 1. The text of the functions appears in Listings 3-13, 3-14, and 3-15.

Listing 3-13 The setcenter() function

```
void setcenter()
// Set center joystick coordinates
{
  while (!readjbutton(JBUTTON1)); // Loop until joystick
                                  //  button pressed
  xcent=readstick(JOY_X);         // Get x coordinate
  ycent=readstick(JOY_Y);         // Get y coordinate
  while (readjbutton(JBUTTON1));  // Loop until button
                                  //  released
}
```

Listing 3-14 The setmin() function

```
void setmin()
// Set minimum joystick coordinates
{
  while (!readjbutton(JBUTTON1)); // Loop until joystick
                                  //  button pressed
  xmin=readstick(JOY_X);          // Get x coordinate
  ymin=readstick(JOY_Y);          // Get y coordinate
  while (readjbutton(JBUTTON1));  // Loop until button
                                  //  released
}
```



Listing 3-15 The setmax() function

```
void setmax()
// Set maximum joystick coordinates
{
  while (!readjbutton(JBUTTON1)); // Loop until joystick
                                  //  button pressed
  xmax=readstick(JOY_X);          // Get x coordinate
  ymax=readstick(JOY_Y);          // Get y coordinate
```

```
  while (readjbutton(JBUTTON1));  // Loop until button
                                  //  released
}
```

The getevent() Function

The main body of the event manager is the *getevent()* function. This function takes two parameters: an integer bitmask specifying which types of events are desired, and an event structure in which the current events (as it were) will be returned. The bitmask consists of the constant values KEYBOARD_EVENTS, MOUSE_EVENTS, and JOYSTICK_EVENTS (defined in EVNTMGR1.H) added together as desired. For instance, to obtain keyboard events and joystick events, you would call the event manager like this:

```
getevent(KEYBOARD_EVENTS+JOYSTICK_EVENTS, &events);
```

where *&events* is a pointer to a variable of type *event_structure*.

The *getevent()* function begins its work by clearing any previous events out of the *event_structure* variable that it is to return events in:

```
  // Clear any events in structure:

  events->go_forward=0;
  events->go_back=0;
  events->go_left=0;
  events->go_right=0;
  events->quit_game=0;
```

If joystick events have been requested, it then calls the low-level joystick functions to determine if any events have occurred. If they have, it sets the appropriate fields in the event structure:

```
  // If joystick events requested....
  if (event_mask & JOYSTICK_EVENTS) {
    if (readstick(JOY_Y)<(xcent-4)) events->go_forward=1;
    if (readstick(JOY_Y)>(xcent+10)) events->go_back=1;
    if (readstick(JOY_X)<(xcent-4)) events->go_left=1;
    if (readstick(JOY_X)>(xcent+10)) events->go_right=1;
    if (readjbutton(JBUTTON1)) events->quit_game=1;
}
```

If mouse events have been requested, then the event manager calls the low-level mouse functions to determine if any events have occurred. If they have, it sets the appropriate fields in the event structure:

```
  // If mouse events requested....
  if (event_mask & MOUSE_EVENTS) {
    relpos(&x,&y);  // Read relative mouse position
```

```
    if (readstick(JOY_Y)<(xcent-4)) events->go_forward=1;
    if (readstick(JOY_Y)>(xcent+10)) events->go_back=1;
    if (readstick(JOY_X)<(xcent-4)) events->go_left=1;
    if (readstick(JOY_X)>(xcent+10)) events->go_right=1;
    if (readjbutton(JBUTTON1)) events->quit_game=1;
}
```

If mouse events have been requested, then the event manager calls the low-level mouse functions to determine if any events have occurred. If they have, it sets the appropriate fields in the event structure:

```
  // If mouse events requested....
  if (event_mask & MOUSE_EVENTS) {
    relpos(&x,&y);  // Read relative mouse position
    if (y<-5) events->go_forward=1;
    if (y>5) events->go_back=1;
    if (x<-20) events->go_left=1;
    if (x>20) events->go_right=1;
    int b=readmbutton(); // Read mouse button
    if (b&MBUTTON1) events->quit_game=1;
}
```

Finally, if keyboard events have been requested, the event manager checks the *keybuffer* array to determine if any events have occurred. If they have, it sets the appropriate fields in the event structure:

```
  // If keyboard events requested....
  if (event_mask & KEYBOARD_EVENTS) {
    if (keybuffer[FORWARDKEY]) events->go_forward=1;
    if (keybuffer[BACKKEY]) events->go_back=1;
    if (keybuffer[LEFTKEY]) events->go_left=1;
    if (keybuffer[RIGHTKEY]) events->go_right=1;
    if (keybuffer[QUITKEY]) events->quit_game=1;
  }
```

The constants used here for the keyboard scan codes are defined in the EVNTMGR1.H file. FORWARDKEY is the scan code for the ↑ key, BACKKEY is the scan code for the ↓ key, LEFTKEY is the scan code for the ← key, RIGHTKEY is the scan code for the → key, and QUITKEY is the scan code for the ESC key. However, the definitions of these constants can be changed in the header file so that any set of five keys can be used to trigger these events. For instance, the scan code for the "8" key on the keypad could be assigned to FORWARDKEY, and so forth. By using variables in place of these constants, a game program could be designed in such a way that the player could redefine at any time the keys used to trigger the events, perhaps from a special configuration screen.

The complete text of the *getevent()* function appears in Listing 3-16.

```
  events->go_right=0;
  events->quit_game=0;

  // If joystick events requested....
  if (event_mask & JOYSTICK_EVENTS) {
    if (readstick(JOY_Y)<(xcent-4)) events->go_forward=1;
    if (readstick(JOY_Y)>(xcent+10)) events->go_back=1;
    if (readstick(JOY_X)<(xcent-4)) events->go_left=1;
    if (readstick(JOY_X)>(xcent+10)) events->go_right=1;
    if (readjbutton(JBUTTON1)) events->quit_game=1;
}
  // If mouse events requested....
  if (event_mask & MOUSE_EVENTS) {
    relpos(&x,&y);   // Read relative mouse position
    if (y<-5) events->go_forward=1;
    if (y>5) events->go_back=1;
    if (x<-20) events->go_left=1;
    if (x>20) events->go_right=1;
    int b=readmbutton(); // Read mouse button
    if (b&MBUTTON1) events->quit_game=1;
}
  // If keyboard events requested....
  if (event_mask & KEYBOARD_EVENTS) {
    if (keybuffer[FORWARDKEY]) events->go_forward=1;
    if (keybuffer[BACKKEY]) events->go_back=1;
    if (keybuffer[LEFTKEY]) events->go_left=1;
    if (keybuffer[RIGHTKEY]) events->go_right=1;
    if (keybuffer[QUITKEY]) events->quit_game=1;
  }
}
```

Animating the Maze

We now have almost all the tools we need for an animated tour of the maze we created a still picture of in the last chapter. You might think that the animation itself would require a great deal of additional code, but that's not true. Animation is simply a matter of drawing a sequence of successive images on the computer display, erasing one image, and then drawing the next image on top of it. In the last chapter we developed the code that draws the image; in this chapter we developed the code that allows the user to interact with that image. Now all we have to do is put them together and we have a game. Well, almost....

We'll reuse much of the same code we used in our maze-drawing program in the last chapter. We'll add some instructions to that code, to initialize the event manager and calibrate the joystick:

```
  // Initialize event manager:
  init_events();
```

```
// Calibrate the user's joystick:
  if (WHICH_EVENTS & JOYSTICK_EVENTS) {
    printf("\nCenter your joystick and press button ");
    printf("one.\n");
    setcenter(); // Calibrate the center position
    printf("Move your joystick to the upper lefthand ");
    printf("corner and press button one.\n");
    setmin();    // Calibrate the minimum position
    printf("Move your joystick to the lower righthand ");
    printf("corner and press button one.\n");
    setmax();    // Calibrate the maximum position
  }
```

We'll need a method for timing the frames of the animation, so that they don't flash by too quickly on faster machines (or too slowly on slower machines). We'll do this by calling the C/C++ library function *clock()*, defined in the header file TIME.H. This function returns the number of *ticks* that have transpired since the program has started running. How long is a *tick?* That may vary from computer to computer, so the C/C++ compiler maintains a special macro called CLK_TCK that can be used to obtain the number of clock ticks in a second. To determine how many ticks we wish to wait between animation frames, we'll establish a constant called FRAMES_PER_SECOND, which we'll initially define as being equal to 6:

```
#define   FRAMES_PER_SECOND 6
```

Then we'll divide CLK_TCK by frames per second to obtain the number of ticks per frame:

```
// Set number of ticks per frame:
ticks_per_frame = CLK_TCK / FRAMES_PER_SECOND;
```

So that we'll know just how much time the first frame of the animation takes, we'll record the current tick count in the variable *last_frame*. This variable must be of type *clock_t*, a special data type defined in the TIME.H header file (which is usually just a long integer):

```
// Initialize the frame timer:
clock_t lastframe=clock();
```

Since our animation loop will continue until a quit_game event occurs, we need to be sure that the quit_game field in the *events* structure — the structure of *event_type* that we'll be using to pass events from the event manager — is set to 0, so that the animation loop will execute at least once:

```
// Make sure we get at least one frame
// into the maze:
events.quit_game=0;
```

Then we start executing an animation loop that will continue executing as long as no quit_game events occur:

```
// Let's go for a walk in the maze:
while(!events.quit_game) {
```

We then call the *drawmaze()* function that we developed in the last chapter to draw our initial position in the maze (which is defined by the same set of variables established in the last chapter):

```
// Draw the maze in screen buffer:
drawmaze(screen_buffer);
```

So that the image of the maze is not whisked too quickly from the display, we must pause until a number of clock ticks has passed equal to the *ticks_per_frame* value that we established a few instructions ago:

```
// Pause until time for next frame:
while ((clock()-lastframe)<ticks_per_frame);
```

We must then reset the value of the *lastframe* variable to the current tick count:

```
// Start timing another frame:
lastframe=clock();
```

If any movement events have occurred, we'll need to change our position within the maze. So we call the event manager to check for input:

```
// Check for input events:
getevent(WHICH_EVENTS,&events);
```

Should a *go_forward* event occur, we'll want to move to the next position in the maze along our current heading. You'll recall from the last chapter that our position within the maze is contained in the variable *pos*, which is of type *xy* (that is, it consists of an *x* field and a *y* field) and our heading is contained in the integer variable *direction*. The forward increments along that heading are contained in the array element *increment[direction]*, which is also of type *xy*. By adding the value of *increment[direction].x* to our x position and *increment[direction].y* to our y position, we can determine our next position along the current heading. However, we'll first assign this position to a temporary variable of type *xy* called *newpos*, so that we can test to see if the maze square at this position is empty before we change our actual position. If the next square is not empty, we can't move into it, because there's a wall in the way:

```
// Do we want to move forward?
  if (events.go_forward) {
    newpos.x=pos.x+increment[direction].x;
    newpos.y=pos.y+increment[direction].y;
```

```
    if (!maze[newpos.x][newpos.y]) {
       pos.x=newpos.x;
       pos.y=newpos.y;
    }
  }
```

If a go_back event has occurred, we'll want to do the same thing in the direction opposite to our current heading. However, we'll make these instructions part of an *else,* so we can't move both forward and backward should the player be pressing both keys simultaneously:

```
// ...or do we want to move backward?
  else if (events.go_back) {
    newpos.x=pos.x-increment[direction].x;
    newpos.y=pos.y-increment[direction].y;
    if (!maze[newpos.x][newpos.y]) {
       pos.x=newpos.x;
       pos.y=newpos.y;
    }
  }
```

We'll also want to check to see if a go_right or go_left event has occurred. If so, we'll rotate our heading clockwise or counterclockwise, respectively:

```
  // Do we want to turn left?
  if (events.go_left) {
    --direction;
    if (direction<0) direction=3;
  }

  // ...or do we want to turn right?
  else if (events.go_right) {
    direction++;
    if (direction>3) direction=0;
  }
}
```

And that's the end of the animation loop. Everything is now set up for the next frame of the animation, which will be drawn by the call to the *drawmaze()* function that we placed at the top of the loop.

When the loop terminates, we'll need to deinstall the event manager:

```
// Terminate event manager:
  end_events();
```

Fixing the Flicker

What sort of animation will this loop produce? Alas, not *quite* the type we want. The program GOMAZE1.EXE on the disk is written exactly as we have described

here. Run it and you'll see immediately what's missing. You can move through the maze by pressing the keys on the keyboard, but the animation of the maze is marred by an irritating flicker. What causes this flicker? You're watching the image of the maze being drawn — and erased — six times a second. It makes sense that such an image would flicker, because it's constantly vanishing and reappearing.

How can we fix this flicker? By drawing and erasing the maze in an offscreen buffer — and only moving the image onto the screen once it's complete. That way, one complete image will rapidly replace another complete image on the video display, with no flicker in sight. To accomplish the task of moving the image from the buffer to the display, we'll need a fast assembly language function to move a rectangular segment of memory into video RAM. Why not move the whole contents of the buffer into video RAM? Because it takes too long. We only need to move that section of the screen that actually contains the maze. We'll use the *putwindow()* function, developed in the last chapter, for that task.

The GOMAZE Program

The final version of the *main()* function of the GOMAZE.CPP program appears in Listing 3-17.

Listing 3-17 The main() function of the GOMAZE.CPP program

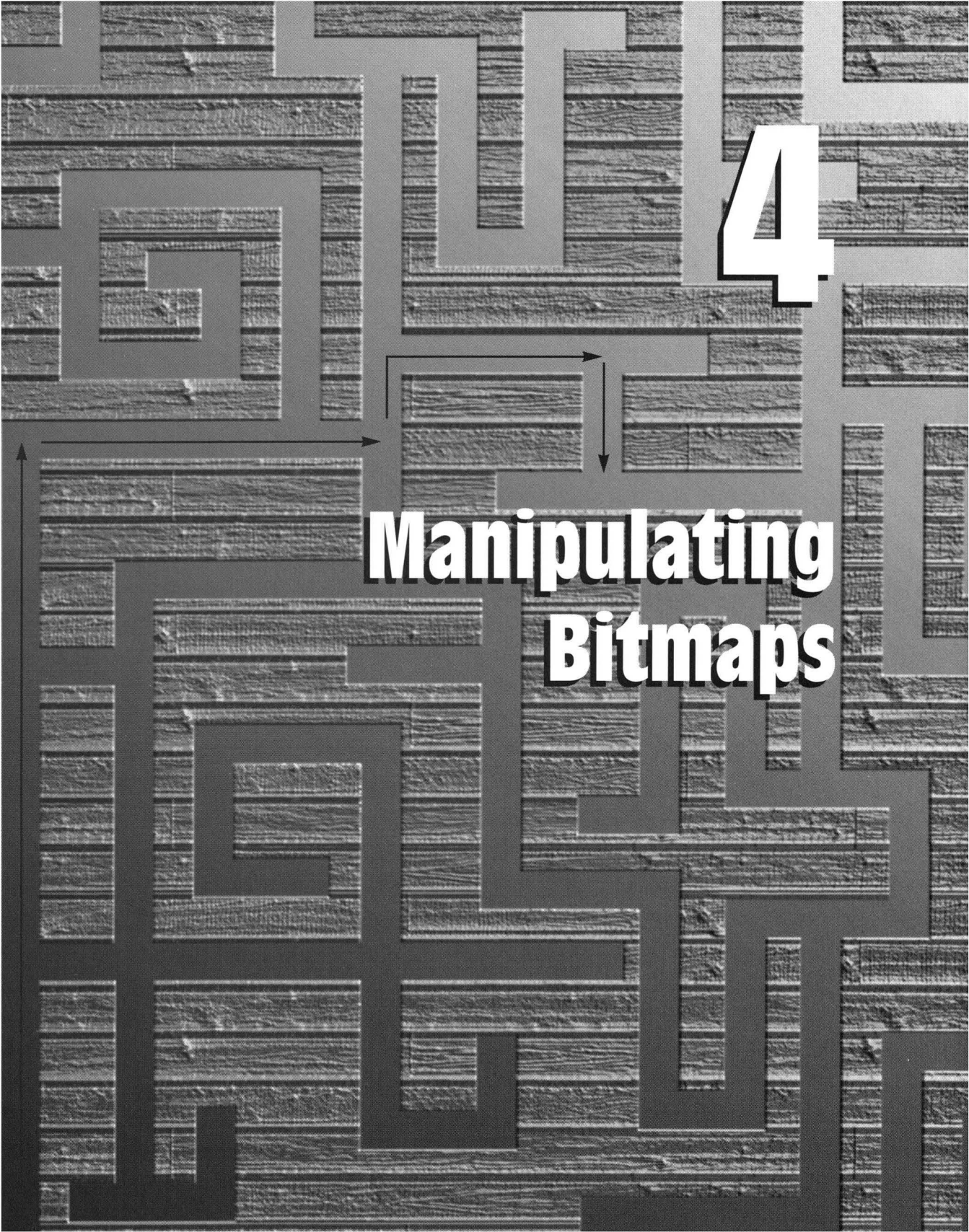
```
void main(void)
{

  int scan;
  event_struct events;
  struct xy newpos;

  // Initialize event manager:
  init_events();

  // Calibrate the user's joystick:
  if (WHICH_EVENTS & JOYSTICK_EVENTS) {
    printf("\nCenter your joystick and press button ");
    printf("one.\n");
    setcenter(); // Calibrate the center position
    printf("Move your joystick to the upper lefthand ");
    printf("corner and press button one.\n");
    setmin();    // Calibrate the minimum position
    printf("Move your joystick to the lower righthand ");
    printf("corner and press button one.\n");
```

continued on next page

continued from previous page

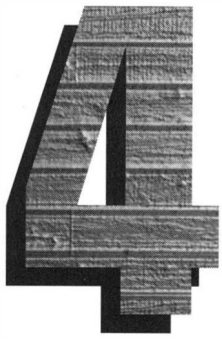
```
    setmax();    // Calibrate the maximum position
  }

  // Create pointer to video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Create offscreen video buffer:
  char far *screen_buffer=new unsigned char [64000];

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Put display in mode 13h:
  setmode(0x13);

  // Clear display:
  cls(screen_buffer);

  // Draw window on display:
  drawbox(screen_buffer);

  // Set number of ticks per frame:
  ticks_per_frame = CLK_TCK / FRAMES_PER_SECOND;

  // Initialize the frame timer:
  clock_t lastframe=clock();

  // Make sure we get at least one frame
  // into the maze:
  events.quit_game=0;

  // Let's go for a walk in the maze:
  while(!events.quit_game) {

    // Draw the maze in screen buffer:
    drawmaze(screen_buffer);

    // Move screen buffer to screen:
    putwindow(0,0,320,200,screen_buffer);

    // Pause until time for next frame:
    while ((clock()-lastframe)<ticks_per_frame);

    // Start timing another frame:
    lastframe=clock();

    // Check for input events:
    getevent(WHICH_EVENTS,&events);

    // Move viewer according to input events:

    // Do we want to move forward?
    if (events.go_forward) {
```

```
      newpos.x=pos.x+increment[direction].x;
      newpos.y=pos.y+increment[direction].y;
      if (!maze[newpos.x][newpos.y]) {
         pos.x=newpos.x;
         pos.y=newpos.y;
      }
    }

    // ...or do we want to move backward?
    else if (events.go_back) {
      newpos.x=pos.x-increment[direction].x;
      newpos.y=pos.y-increment[direction].y;
      if (!maze[newpos.x][newpos.y]) {
         pos.x=newpos.x;
         pos.y=newpos.y;
      }
    }

    // Do we want to turn left?
    if (events.go_left) {
      --direction;
      if (direction<0) direction=3;
    }

    // ...or do we want to turn right?
    else if (events.go_right) {
      direction++;
      if (direction>3) direction=0;
    }
  }

  // Terminate event manager:
  end_events();

  // Release memory
  delete screen_buffer

  // Restore old video mode:
  setmode(oldmode);
}
```



4

Manipulating Bitmaps



Let's be honest. The wireframe maze animation that we created in the last two chapters, while it illustrated principles of graphics output and device input that *could* be used in a state-of-the-art game program, was not exactly of commercial quality itself. To the best of my knowledge, the last major commercial game that featured wireframe maze graphics was Wizardry V, published in 1989. And that game looked like something from the distant past even at the time of its publication. More recent installments in the Wizardry series have made quantum leaps beyond the simple wireframe mazes of the first five entries. As entertaining as the early Wizardry games were (and still are), no serious game programmer can get away with this sort of simplistic maze animation in the 1990s.

So, in this chapter, we're going to make a quantum leap of our own. We'll throw the wireframe maze graphics aside and build the core of a bitmapped maze engine similar to those used in such up-to-the-minute games as Eye of the Beholder 1, 2, & 3 and Wizardry VII: Crusaders of the Dark Savant. This engine will use many of the programming principles that we've discussed in the last two chapters, but what you'll see on the screen will no longer be white lines on a black screen. Instead, you'll see a realistic, full-color animated maze that could have leaped right out of one of the games on the current software bestseller list. As a bonus, we'll show you how to generate new graphics for this maze engine using software that can be obtained quite inexpensively or even (if you play your cards right) for free.

These bitmapped maze graphics will be constructed out of predesigned graphic bitmaps, which (as it happens) we'll be creating using the ray-tracing program POVRAY. You can use other methods for constructing bitmapped maze graphics — so-called "paint" programs like Electronic Arts' Deluxe Paint are an excellent way to create bitmaps, if you have the requisite artistic talent (or can afford to hire the services of someone who does). Nevertheless, a ray tracer is a great way to throw together quick-and-dirty graphics of surprising beauty and realism, especially if you are less than deft at painting pixels yourself.

Before we can use the bitmaps created by POVRAY to create a maze, though, we must have some method of storing these bitmaps on disk and loading them into the computer. So in the first portion of this chapter, we'll concentrate not on creating bitmaps, but on storing and retrieving them from disk.

What Is a Bitmap?

We saw in chapter 2 how computers like the IBM PC and its clones store the current video image in memory as a sequence of byte values representing either characters or pixels on the display. In mode 13h, each byte in video memory represents the color of a corresponding pixel on the screen. Because these bytes of data, like all numbers held in the electronic circuitry of a computer, are made up of individual binary digits, or bits, it's common to refer to this method of representing pixels as a bitmap. In the case of mode 13h, it might seem more appropriate to use the term "bytemap," but the term bitmap goes back to the days when it was possible to describe the state of pixels on the typical video display using only 1 or 2 bits within a byte. This is still the case with the PC's two- and four-color graphics modes, but as computer graphics move into the era of 256-color images and beyond, bitmaps made of individual bits are becoming increasingly less common. Now 4 or 8 or even 24 bits per pixel may be required.

Bitmaps aren't always stored in video memory. In the last chapter, we stored the bitmap representing the next frame of animation in an offscreen buffer in video RAM until we were ready to move it onto the video display. For longer-term storage, it's quite possible to store bitmapped images in a disk file on a hard drive or CD ROM. In fact, most computer games store their graphics in precisely this way. If you've ever wondered why the typical computer game takes up several megabytes of space on your disk, it's probably because the game includes a vast library of 256-color bitmapped images stored in the same directory with the game, or in one of the subdirectories off of that directory. (Digitized sound files are also a popular method of burning up large amounts of disk storage, especially in the case of CD ROM-based games, which typically have about half a gigabyte

of storage available to devote to this and other types of data.) When the game needs to display these graphics, the contents of the files are read from the disk and copied into video memory, often by way of one or more off-screen buffers.

What form do 256-color graphics take when stored on a disk? In some cases they are stored in exactly the same way that they are normally represented in video memory, as a sequence of byte values representing the pixels on the display, perhaps with a list of 256 color descriptors (each consisting of 3 bytes of color data, as described in chapter 2) appended to the end of the file so that the image's palette can be reconstructed when it's displayed. This is not, however, a particularly efficient way of storing an image. Quite often, the graphics are stored in compressed format, with as much redundant information as possible squeezed out of them. If you think the games on your hard drive are taking up a lot of space *now,* you should see how much space they'd take up if their graphics weren't compressed!

In this chapter we'll be constructing images of a maze out of predrawn bitmaps. We'll need to store those bitmaps on the disk before we can use them. It's important that we give some thought about how we're going to do this, so let's consider some possible formats in which we could store the files.

Graphics File Formats

There are probably an infinite number of different ways in which graphics bitmaps can be stored in a disk file. Fortunately, we don't have to reinvent the wheel. There are a number of standard file formats available for us to use. Let's look at a few of them and see what sort of strengths and weaknesses they have.

The *PCX* format was developed in the mid-1980s for a program called PC Paintbrush, published by ZSoft. (The format is also referred to as PC Paintbrush format or ZSoft format.) This is probably the single most widely supported graphics file format for the PC. Although the data compression algorithm used to squeeze the redundancy out of the files isn't the most efficient, it's very easy (and fast) to decompress. PCX files are identified by a file extension of .PCX, as in IMAGE.PCX.

The *Graphics Interchange File (GIF)* format was developed by the CompuServe Information Service as a means of viewing graphics while logged on to the service via modem. Although files in this format were originally intended primarily for on-line viewing, the GIF (pronounced "jiff") format has also become quite popular as a means of distributing graphics images from computer to computer. The compression scheme is more effective than the one used in PCX files, but it's also harder (and somewhat slower) to decode. GIF files are identified by an extension of .GIF, as in IMAGE.GIF.

The *Targa* file format *(TGA)* is named after a line of graphics add-on boards sold by the Targa company. Because the most popular Targa boards are those that offer 24-bit color (that is, bitmaps that represent each pixel with 3 bytes of data and therefore can contain up to 16 million different colors at one time), most Targa files contain 24-bit images. These images are stored in so-called direct color format, in which each pixel is represented by a 3-byte color descriptor, with 8-bit values for the red, blue, and green intensities of the color. These files contain no palette, since any palette information would be redundant, the color information being stored in the bitmap itself. However, the Targa format also allows for the storage of 8-bit (256-color) images with an attached palette. Targa files are often uncompressed. This format is quite popular, in part because of the simplicity of the uncompressed Targa format and in part because it's the most widely known method of storing 24-bit images (though uncompressed 24-bit image files can be quite large). Targa files are identified by an extension of .TGA, as in IMAGE.TGA.

A large number of other formats are available, such as the Windows *BMP* (short for bitmap) format. If you'd like a detailed survey of the most common ones, you might want to take a look at the book *Graphics File Formats* (Windcrest/McGraw Hill, 1992) by David C. Kay and John R. Levine. No graphics programmer should be without at least one reference to the standard PC file formats, and this is a particularly useful one.

In this chapter we'll use files in two different formats, the uncompressed 8-bit TGA format and the PCX format. I've chosen the latter because it's supported by a wide variety of graphics utilities and it provides an easily decoded compression scheme. I've chosen the former because it is simple to implement and happens to work well with the graphics utilities that I used to create the maze bitmaps that we'll use in the maze program that we develop in this chapter. Let's look at these two formats in turn.

The 8-Bit Targa Format

The 256-color, 8-bit Targa file format is actually rather rare, having been eclipsed long ago by the 24-bit version. However, in creating the graphics for the maze program in this book, I used the public domain Persistence of Vision ray tracer (better known simply as POV or POVRAY) and the public domain Piclab graphics conversion utility, both of which work well with Targa files. POVRAY stores the images it creates in the 24-bit Targa format. Piclab is easily used to downscale these 16 million color images to 256-color images, and will store the converted images as 8-bit Targa files, which can be viewed on PCs without 24-bit graphic adapters. (It will also store the converted images as GIF files, but the GIF

compression scheme is more complex to decode.) I decided that it would make my job a great deal easier if I included routines in the bitmapped maze program to support the 8-bit Targa format.

Like almost all types of graphics files, Targa files begin with a header containing information about the image or images that the file contains. (Some graphics file formats allow more than one image to be stored in a file, though none of the files on the disk that accompanies this book are stored that way.) We'll need some of the information in this header, so the first thing we'll want to do is read it off the disk. The easiest and most elegant way to do this is to declare a data structure with precisely the same fields as the header and read the header from the file using the C/C++ *read()* function. The definition for the Targa header appears in Listing 4-1.

Listing 4-1 The tga_header structure

```
struct tga_header {
  BYTE ID_length;              // Length of ID field
  BYTE color_map_type;         // 0=direct color, else palette
  BYTE image_type;             // Type of Targa image
  WORD first_color_map_entry;  // First palette color
  WORD color_map_length;       // How long is palette?
  BYTE color_map_entry_size;   // How long is color descriptor?
  WORD image_x_origin;         // Upper left corner X
  WORD image_y_origin;         // Upper left corner Y
  WORD image_width;            // Width in pixels
  WORD image_height;           // Height in pixels
  BYTE bits_per_pixel;         // How many bits describe pixel?
  BYTE image_descriptor_bits;  // Description of image
};
```

The data types BYTE and WORD are aliases for *unsigned char* and *unsigned int*, which I've defined along with this header in the file TARGA.H in an attempt to make the code less cluttered.

This header is followed in turn by an image ID, which is an optional string of characters identifying the image — a name, if you will. The length of this string is defined by the *ID_length* field in the header. Not all Targa files contain an image ID, so this byte will often be equal to 0. Nonetheless, we'll have to check for this when loading a Targa file, so that we can handle the image ID if it's there.

The image ID is followed by the color map, which is roughly equivalent to the palette that we discussed in chapter 2. As with the image ID, not all Targa files contain a color map. In fact, since the popular Targa 24-bit format uses a direct color bitmap rather than a palette-mapped structure, the color map is rather rare in

Targa files. Nonetheless, we'll be using color-mapped Targa files in a couple of the programs that we create in this chapter, so we'll need to deal with the color map. Although the color map can take several forms, we'll deal with 24-bit color maps, identified by a value of 24 in the *color_map_entry_size* field of the header. These color maps are similar but not identical to VGA 256-color palettes, so we'll need to do some converting when we load the color map. More about that in a moment.

Following the color map is the bitmapped image itself. Like the color maps, this bitmap can take several different forms. In this chapter we'll deal exclusively with 8-bit images, identified by a value of 8 in the *bits_per_pixel* field of the header. The bitmap for an 8-bit uncompressed Targa image is identical to the bitmap used in VGA mode 13h: a sequence of bytes representing pixels in which each byte corresponds to one of the 256 color descriptors in the color map. We can read the image data directly into an array, where we can store it prior to putting the bitmap (or portions of the bitmap) on the display.

That's pretty much all there is to an 8-bit uncompressed Targa file. Now let's look at some code for loading Targa files into memory.

An 8-Bit Targa Loader

The 8-bit uncompressed Targa format is so simple that we really only need a single function to load the files into memory. We'll call that function *loadTGA()*. The letters TGA refer to the file extension used on all types of Targa files. The prototype for this function, which we'll place in the header file TARGA.H, is

```
int loadTGA(char far *filename,tga_struct *tga)
```

The parameter *filename* is a string of characters containing the pathname of the Targa file to be loaded, relative to the current directory. The parameter *tga* is a pointer to a variable of type *tga_struct*, which is defined in the TARGA.H header file as

```
struct tga_struct {
  tga_header header;
  char far *ID;
  BYTE far *image;
  BYTE far *color_map;
};
```

This structure has fields in it equivalent to all parts of a Targa file. The header is actually stored within the structure in a field of type *tga_header*, which we defined back in Listing 4-1. The other fields are pointers to the ID string, the image bitmap, and the color map, respectively. We'll need to allocate space for these structures from the heap before loading them off the disk.

Before we can do this, however, we need to load the header from the disk, because it contains important information about the file. If we were writing a

full-service Targa loader that could handle all types of Targa files, we'd use the header information to determine whether to load a direct color or palette-mapped image, whether to load a color map from the file, what format the color map and bitmap were to take, and so forth. However, since we'll only be loading 8-bit palette-mapped uncompressed Targa files, we'll be using the header primarily to decide whether the file that we have been asked to load is indeed such a file. If it isn't, we'll abort the *loadTGA()* function and return an error message to the calling function.

To open the Targa file that we've been requested to load, we first call the C/C++ *open()* function:

```
if ((infile=open(filename,O_BINARY)) == -1) return(-1);
```

The *open()* function, called with these parameters, will search for a file with the path name that was passed to the *loadTGA()* function in the *filename* parameter. If found, it will treat this file as a binary file — that is, as a file made up of unsigned byte values that don't represent ASCII characters. If the file isn't found, the *open()* function returns a value of -1. In that event the *loadTGA()* function will do the same, aborting the function and returning a value of -1 to the calling function, indicating that an error has occurred. (We'll use the -1 value to represent all types of errors encountered by this function.)

Now that the file is open, we'll use the *lseek()* function to place the DOS file pointer at the beginning of the file (which is where we want to begin reading) and the *read()* function to read the header from the file into the *header* field of the *tga* structure:

```
lseek(infile,0L,SEEK_SET);
read(infile,&(tga->header),sizeof(tga_header));
```

We pass both functions the file handle returned by the *open()* function, which is contained in the variable *infile*. The *read()* function also requires a pointer to the header field of the *tga* variable and the number of bytes to read into the structure, which we've determined by calling the *sizeof()* function. This standard library function returns the size of a variable structure, which in this instance corresponds to the number of bytes of data that we wish to read into it.

Once we've read the header data into the *header* field of the *tga* variable, we can access fields within the header the way we'd access any other variables within a structure. We'll use this capability to make sure that the image in the Targa file is of the type that we're expecting. If it isn't, we abort with an error code:

```
if (tga->header.image_type != 1) return(-1);
if (tga->header.image_width> 320) return(-1);
if (tga->header.image_height > 200) return(-1);
if (tga->header.bits_per_pixel != 8) return(-1);
```

```
if (tga->header.color_map_entry_size != 24) return(-1);
```

These instructions check to see if the image is of type 1 (which is an uncompressed color-mapped file); whether the width is greater than 320 (in which case we can't use it on the mode 13h display); whether the height is greater than 200 (same problem); whether the number of bits per pixel is something other than 8; and whether the color map contains something other than 24-bit color descriptors. If the file passes all these checks — as the images that we're going to be using in this chapter will — we can begin processing the body of the file. First we'll load the ID string, in case the calling function is interested in doing something with it:

```
if (tga->header.ID_length>0) {
  tga->ID=new char[tga->header.ID_length];
  read(infile,tga->ID,tga->header.ID_length);
}
```

These instructions first check to see if the file actually contains an ID string by looking at the *ID_length* field in the header. If this field is greater than 0, we call the C++ *new* function, which allocates a chunk of memory equivalent to the value of the *ID_length* field and returns *char* * type pointer to the chunk. We then read the ID string into this chunk of memory with the *read()* function.

Before we can load the color map, we must allocate a 768-byte section of memory to store it in:

```
if ((tga->color_map=new unsigned char[3*256])==NULL)
  return(-1);
```

If the memory allocation fails, we return to the calling function with an error code. (The expression 3*256 is used instead of 768 to remind us that we're loading 256 color descriptors that are each 3 bytes long — and because I find it easier to remember the number 256, which is commonly found in computer programs, than to remember the number 768.) Otherwise, we proceed to load the color map, translating it as we do so into a format acceptable to the VGA 256-color palette-setting function:

```
for (int entry=0; entry<256; entry++) {
  for (int color=2; color>=0; --color) {
    read(infile,&(tga->color_map[entry*3+color]),1);
    tga->color_map[entry*3+color] >>= 2;
  }
}
```

Exactly what sort of translation are we performing here? A VGA color descriptor consists of three values in the range 0 to 63, one each for the red, blue, and green intensities of the color. Only 6 bits are necessary to store values in this range. So, although we allot an 8-bit byte for each of these values when we pass them to the VGA 256-color palette-setting function, the two leftmost bits of those values

aren't used. The Targa color descriptors, by contrast, are values in the range 0 to 255, so they use all 8 bits of the bytes in the descriptor. To translate them to VGA descriptors, we must shift the binary digits of these values two positions to the right, using the C/C++ >> operator. (Actually, we use the >>= operator, which shifts the values and reassigns them to the variables in which they were already contained.) In addition, the Targa color descriptor values are in the order red, green, blue, while the VGA color descriptors should be in the order blue, green, red. Therefore we have to reverse the order of the descriptors as we load them into the array. That's why the *for()* loop that loads the three color descriptor values counts the index variable *color* backward from 2 to 0, storing the values in the reverse order from that in which they are read from the disk.

Since the image data is already stored in exactly the format we need for a mode 13h bitmap, loading the image is a simple matter of calculating the number of bytes it contains (by multiplying the *image_width* field of the header by the *image_height* field), allocating memory for the bitmap array, and reading the bitmap data from the disk into the array:

```
unsigned int image_size=tga->header.image_width
  * tga->header.image_height;
if ((tga->image=new unsigned char[image_size])==NULL)
  return(-1);
read(infile,tga->image,image_size);
```

Our job is now complete, so we close the Targa file and return to the calling function with a return value of 0, indicating that no errors occurred:

```
close(infile);
return(0);
```

The loadTGA() Function

The complete text of the *loadTGA()* function appears in Listing 4-2.

Listing 4-2 The loadTGA() function

```
static int infile; // Handle for Targa file

int loadTGA(char far *filename,tga_struct *tga)
// Loads an 8-bit uncompressed Targa file into a structure of type
// tga_struct
{

  // Open file, abort if not found:
  if ((infile=open(filename,O_BINARY))==-1) return(-1);
```

continued on next page

continued from previous page

```
// Set file pointer at start of file:
lseek(infile,0L,SEEK_SET);

// Read header into header field:
read(infile,&(tga->header),sizeof(tga_header));

// Is this an uncompressed color-mapped image? If not, abort:
if (tga->header.image_type != 1) return(-1);

// Is the image wider than 320 pixels? If so, abort:
if (tga->header.image_width> 320) return(-1);

// Is the image taller than 200 pixels? If so, abort:
if (tga->header.image_height > 200) return(-1);

// Is it an 8-bit image? If not, abort:
if (tga->header.bits_per_pixel != 8) return(-1);

// Does it have a 24-bit palette? If not, abort:
if (tga->header.color_map_entry_size != 24) return(-1);

// If there's an ID string, load it:
if (tga->header.ID_length>0) {
  tga->ID=new char[tga->header.ID_length];
  read(infile,tga->ID,tga->header.ID_length);
}
   else tga->ID=0
// Allocate memory for palette, abort if not available:
if ((tga->color_map=new unsigned char[3*256])==NULL) {
if (tga->ID) delete tga->ID;
  return(-1);
}
// Load palette, translate for VGA:
for (int entry=0; entry<256; entry++) {
  for (int color=2; color>=0; --color) {
    read(infile,&(tga->color_map[entry*3+color]),1);
    tga->color_map[entry*3+color] >>= 2;
  }
}

// Calculate size of image in bytes:
unsigned int image_size=tga->header.image_width
  * tga->header.image_height;

// Allocate memory for image, abort if not available: {
delete tga->color_map;
if (tga->ID) delete tga->ID;
if ((tga->image=new unsigned char[image_size])==NULL)
  return(-1);
}

// Load image from file:
read(infile,tga->image,image_size);
```

```
  // Close file:
  close(infile);

  // Return without error:
  return(0);
}
```

An 8-Bit Targa Display Program

To demonstrate these functions (and, not incidentally, create a utility that will be useful throughout the remainder of this chapter), let's write a program called TGASHOW that will read 8-bit uncompressed Targa files from the disk and display them in mode 13h. Such a program will need to include the TARGA.H header file at the beginning of any modules that reference the Targa loader functions, and should incorporate TARGA.CPP as part of the project file or makefile used to construct the final code. We'll keep our program simple by putting all the unique code not already incorporated elsewhere in one of our support modules in a single *main()* function. This function will begin with the usual *main()* header, but will reference the standard C/C++ argument parameters so that the user will be able to pass the function the name of a Targa file from the command line:

```
void main(int argc,char *argv[])
```

In case you're not familiar with the argument parameters, the integer parameter *argc* will automatically be set equal to the number of arguments (that is, strings of characters separated by blank spaces) that the user types on the command line when the program is invoked, including the name of the program itself. Thus *argc* will always be equal to at least 1. We want the user to be able to invoke the program with the name of a Targa file as an argument, like this:

```
tgashow image.tga
```

So *argc* should be equal to 2 when the main function begins, since there should be two arguments on the command line (counting the name of the program). We'll check for this at the beginning of the *main()* function and abort the program with an error code if the wrong number of arguments is found:

```
if (argc!=2) { // Are there 2 arguments on command line?
  puts("Wrong number of arguments.\n");
  exit(-1);
}
```

The other argument parameter, **argv[]*, is an array of pointers to the actual character strings that make up the arguments typed on the command line. The first element of the array, *argv[0]*, always points to the name of the program itself. If you ever write a program that needs to know what name it was invoked under,

this is the best way to obtain that information. For our Targa-viewing program, we'll use the second element of this array, which will be pointing to the name of the Targa file that the user wants to view. We can pass this array element to the *loadTGA()* function like this:

```
// Load Targa file. Abort if not found:
if (loadTGA(argv[1],&tga)) exit(1);
```

You'll recall that the *loadTGA()* function returns a value of 0 if it's successful in loading the file, -1 if it isn't. Here we simply look for a non-zero value (allowing us to expand our repertoire of error codes at a later date if we wish) and abort to DOS if one is found. The *exit()* function returns an error code of 1 to DOS, which can be detected by a batch file if one is used to invoke the TGASHOW program. The parameter *tga* to which we pass the *loadTGA()* function a pointer is a variable of type *tga_struct*, defined earlier in the program like this:

```
tga_struct tga;
```

Now that the Targa file has presumably been loaded, we can set mode 13h, clear the screen, and establish a pointer to video memory exactly as we've been doing in earlier programs (so we won't bother to detail the procedure here). The palette can be set to the color map from the Targa file using our *set_palette()* function, like this:

```
// Set palette to Targa color map:
setpalette(tga.color_map);
```

The image itself can be copied directly from the *tga.image* field into video memory, with the aid of our usual *screen* pointer. However, we must take into account the possibility that the Targa image will be smaller than the 320 by 200 mode 13h screen (as all of the Targa images we use in this chapter will be). The information about the size of the image is contained in the *image_width* and *image_height* fields of the header structure, which contain the width and height of the image in pixels. We can use these as the limiting values of a pair of nested *for()* loops, which will work their way across the image in the x and y dimensions, copying pixels as they go:

```
// Copy image to mode 13h display:
for (int y=0; y<tga.header.image_height; y++) {
  for (int x=0; x<tga.header.image_width; x++) {
    screen[y*320+x]=tga.image[y*tga.header.image_width+x];
  }
}
```

The position at which to fetch each pixel from the Targa image array is calculated by multiplying the y position of the pixel by the width of the image and adding in the x position. The position at which to place the pixel on the mode 13h display is similarly calculated by multiplying the y position of the pixel by the

width of the screen (320) and adding in the x position. This will cause images smaller than 320 by 200 to appear in the upper-left corner of the display. Savvy programmers will note that this is a rather slow method of copying data — it wouldn't be difficult to come up with a faster technique — but time isn't exactly of the essence in this program.

The TGASHOW Program

The complete text of TGASHOW.CPP appears in Listing 4-3.

Listing 4-3 The TGASHOW.CPP program

```
// TGASHOW.CPP Version 1.0
// Display a 256-color Targa image on the mode 13h screen
//
// Written by Christopher Lampton
// for Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  "screen.h"
#include  "targa.h"

tga_struct tga;

void main(int argc,char *argv[])
{
  if (argc!=2) { // Are there 2 arguments on command line?
    puts("Wrong number of arguments.\n");
    exit(-1);
  }

  // Load Targa file. Abort if not found:
  if (loadTGA(argv[1],&tga)) exit(1);

  // Create pointer to video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Put display in mode 13h:
  setmode(0x13);

  // Clear display:
  cls(screen);
```

continued on next page

continued from previous page

```
  // Set palette to Targa color map:
  setpalette(tga.color_map);

  // Copy image to mode 13h display:
  for (int y=0; y<tga.header.image_height; y++) {
    for (int x=0; x<tga.header.image_width; x++) {
      screen[y*320+x]=tga.image[y*tga.header.image_width+x];
    }
  }

  // Pause until user presses a key:
  while (!kbhit());

  // Release memory
  delete tga.image;
  delete tga.color_map;
  if (tga.ID) delete tga.ID;

  // Restore old video mode:
  setmode(oldmode);
}
```

There's a copy of this program on the disk that accompanies this book. To give it a test run, CD to the subdirectory called BITMAP in the directory where you placed the files from this book. Then type the following on the command line:

```
tgashow testmaze.tga
```

A 160 by 100 image will appear in the upper-left corner of the mode 13h display. The image, which you can see reproduced here in Figure 4-1, is an example of the maze graphics that we'll be developing in this chapter. More about this later.

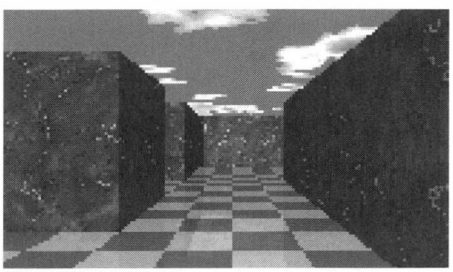
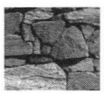


Figure 4-1 The TESTMAZE.TGA image, as displayed by the TGASHOW program. This is a sample of the maze graphics that we'll be developing later in this chapter

The PCX File Format

Before we talk about using predefined bitmaps in animated maze games, we need to discuss one more file format. Although we'll be using the Targa file format for easy compatibility with POVRAY and PicLab, we're going to use the PCX format in all cases where a Targa file isn't strictly necessary. One reason for this is that the PCX format includes data compression, allowing us to store the bitmap data in a smaller amount of space. With state-of-the-art computer games now taking up as much as 35 megabytes of hard drive real estate, the users of our games will thank us for conserving as much of their disk space as we can. (At least, they'll be a little less angry at us if we don't wipe out those last few bytes that they'd intended to devote to spreadsheet data.) Another reason for using PCX files is that many paint programs, such as Deluxe Paint, will create files in this format.

The PCX format, like the Targa format, begins with a file header describing the image stored in the file. Listing 4-4 shows the declaration for the header format structure, which we'll place in the file PCX.H.



Listing 4-4 The PCX_HEADER structure

```
struct pcx_header {
  char manufacturer;    // Always set to 0
  char version;         // Always 5 for 256-color files
  char encoding;        // Always set to 1
  char bits_per_pixel;  // Should be 8 for 256-color files
  int  xmin,ymin;       // Coordinates for top left corner
  int  xmax,ymax;       // Coordinates for bottom right corner
  int  hres;            // Horizontal resolution of image
  int  vres;            // Vertical resolution of image
  char palette16[48];   // EGA palette; not used for
                        //  256-color files
  char reserved;        // Reserved for future use
  char color_planes;    // Color planes
  int  bytes_per_line;  // Number of bytes in 1 line of
                        //  pixels
  int  palette_type;    // Should be 2 for color palette
  char filler[58];      // Nothing but junk
};
```

Several of these fields correspond to the ones that we found of interest in the Targa header, though it isn't always obvious which ones. Oddly, there are no fields that correspond precisely to the *image_width* and *image_height* fields in the Targa header. Fortunately, this information can be calculated from the information in the header like this:

```
int image_width = pcx.header.xmax - pcx.header.xmin +1;
int image_height = pcx.header.ymax - pcx.header.ymin +1;
```

where *pcx* is a variable of type *pcx_struct*. The *filler* field at the bottom of the structure contains no valid data at all, having apparently been included to bulk the structure out to an even 128 bytes. It's conceivable that this portion of the header will be used in future versions of the PCX standard, but for now it can be safely ignored.

The definition for our PCX structure, which is also included in PCX.H, is shown in Listing 4-5.

Listing 4-5 The PCX_STRUCT structure

```
struct pcx_struct {
  pcx_header header;            // Structure for holding the PCX
                                //  header
  unsigned char far *image;     // Pointer to a buffer holding
                                //  the 64,000-byte bitmap
  unsigned char palette[3*256]; // Array holding the 768-
                                //  byte palette
};
```

Aside from the PCX header, the PCX structure contains pointers to a *char* array containing the PCX image bitmap and a second *char* array containing the color descriptors in the palette. These two portions of the PCX file — the image and the palette — follow immediately after the header.

Run-Length Encoding

The PCX image is not stored in the PCX file as a straightforward bitmap, the way the uncompressed Targa image was. Rather, it is stored in compressed format. The type of compression used in PCX files is known as run-length encoding, or RLE for short.

Run-length encoding is not a specific method of bitmap compression, but a generic term for a whole battery of data compression techniques. What all of these techniques have in common is that they store bytes in terms of runs. A run is a sequence of bytes that all have the same value. This occurs commonly in bitmaps, where several pixels of the same color often appear in a row. In an

uncompressed 256-color bitmap, these rows of identically colored pixels are stored as runs of bytes representing the pixels. If there are 30 pixels in a row, all in palette color 117, then the bitmap will contain 30 bytes in a row with a value of 117.

This is — quite literally — redundant. If you've seen one pixel with a value of 117, then you've seen them all — and we don't need to store them all. Instead of storing 30 sequential bytes all with a value of 117, we can store the number 30 followed by the number 117. Such a pair of numbers is called a run-length pair; the first number in such a pair always gives the number of bytes in the run and the second number gives the numerical value of those bytes. Storing a run of 30 bytes in this manner would save us 28 bytes of storage, which is well worth the effort if disk space is at a premium (which it usually is).

When there are several bytes in a row with *different* values, this sort of run-length encoding can be quite inefficient. Each byte would need to be stored as a run 1 byte long — that is, as a run-length pair in which the first byte is a 1 and the second byte is the original value — which would double the storage required for those bytes. The ideal run-length scheme would include a way of turning RLE on when it's needed and off when it isn't.

The PCX RLE scheme does this by letting byte values in the range 192 though 255 represent run-length values in the range 0 to 63. When a byte in this range is detected, the run-length decoder — the program that decompresses the RLE-encoded data — subtracts 192 from it and repeats the following byte value that many times. For instance, if the number 207 is encountered followed by the number 16, the decoder translates this into a run of 15 (207 minus 192) bytes with a value of 16, like this:

```
16,16,16,16,16,16,16,16,16,16,16,16,16,16,16
```

How does the PCX RLE scheme encode individual values in the range 192 to 255 so that they won't be mistaken for run-length values? Alas, it must encode them as runs of a single byte in length — that is, as the number 193 followed by the value to be encoded. For instance, an individual byte with a value of 239 would be encoded as the number 193 followed by the number 239. This is a flaw in the PCX scheme that prevents it from achieving the maximum possible run-length compression — all those extra run-length values make the data larger than it has to be — but doesn't interfere in any way with the decoding of the data. In fact, it makes decoding simpler because we can decode these single-byte runs the same way that all other runs are encoded.

The loadPCX() Function

To decompress an RLE-encoded PCX bitmap, we're going to write a function called *loadPCX()*. Because the RLE-encoding makes the PCX file format a bit more complicated than the uncompressed Targa format, we'll give this function a pair of subsidiary functions that it can call to load the image and the palette. We'll call these subsidiary functions *load_image()* and *load_palette()*, respectively.

The prototype for the *loadPCX()* function looks like this:

```
int loadPCX(char far *filename,pcx_struct *pcx)
```

The parameters are essentially the same as the ones passed to the *loadTGA()* function that we described earlier in this chapter. The *filename* parameter is a pointer to an array of type *char* containing the pathname of the PCX file to be viewed, and the *pcx* parameter is a pointer to a structure of type *pcx_struct* into which will be placed the file data.

We then proceed to open the file and read the header data into the *pcx* structure, exactly as we did in the *loadTGA()* function:

```
if ((infile=open(filename,O_BINARY))==-1) return(-1);
lseek(infile,0L,SEEK_SET);
read(infile,&(pcx->header),sizeof(pcx_header));
```

Now that we have the header data in memory, we check to see if the file contains a 320 by 200 or smaller image. If it doesn't, we return to the calling function with an error code:

```
if (pcx->header.xmax - pcx->header.xmin+1 > 320) return(-1);
if (pcx->header.ymax - pcx->header.ymin+1 > 200) return(-1);
```

We also check to make sure that the PCX file is in Version 5 format, since this is the only type of PCX file that can store a 256-color image:

```
if (pcx->header.version != 5) return(-1);
```

If all checks out, we call the *load_image()* and *load_palette()* functions, close the file, and return to the calling function without an error code:

```
if (load_image(infile,pcx)) return(-1);
load_palette(infile,pcx);
close(infile);
return(0);
```

Note that we abort with an error code if the *load_image()* function fails, but don't bother to check for an error code from *load_palette()*. That's because the *load_palette()* function doesn't return error codes, as you'll see in a moment.

The complete text of the *loadPCX()* function appears in Listing 4-6.

Listing 4-6 The loadPCX() function

```
static int infile;

int loadPCX(char far *filename,pcx_struct *pcx)
{
  if ((infile=open(filename,O_BINARY))==-1) return(-1);
  lseek(infile,0L,SEEK_SET);
  read(infile,&(pcx->header),sizeof(pcx_header));
  if (pcx->header.xmax - pcx->header.xmin+1 > 320) return(-1);
  if (pcx->header.ymax - pcx->header.ymin+1 > 200) return(-1);
  if (pcx->header.version != 5) return(-1);
  if (load_image(infile,pcx)) return(-1);
  load_palette(infile,pcx);
  close(infile);
  return(0);
}
```

The load_image() Function

The *load_image()* function is rather complex, since it must decode the run-length encoding scheme used to encode PCX bitmaps. Basically, it operates in two modes, identified by the current value of the integer variable *mode*. This variable can take either of two values, defined at the head of the function as RUN_MODE and BYTE_MODE. When in BYTE_MODE, the *load_image()* function reads the next byte from the PCX file and checks to see if it's in the range 192 to 255. If it isn't in that range, it is pixel data and the function simply outputs the value of the byte to the buffer that holds the decoded bitmap. If it is in that range, the function subtracts 192 from the value, sets the integer variable *bytecount* equal to the result, fetches the next value from the file and stores it in the integer variable *outbyte*, and sets the *mode* variable to RUN_MODE. While in RUN_MODE, the function repeatedly outputs the value of *outbyte* to the decoding buffer until *bytecount* has been decremented to 0, thus decompressing the run of bytes.

Here's the prototype for the *load_image()* function:

```
int load_image(int pcxfile,pcx_struct *pcx);
```

The first parameter, *pcxfile,* is the handle of the PCX file as returned by the *open()* function. The second parameter, *pcx*, is the same pointer to the *pcx_struct* variable that was passed to *loadPCX()* by the calling function. The complete text of the *load_image()* function appears in Listing 4-7.

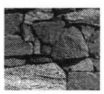
Listing 4-7 The load_image() function

```
int load_image(int pcxfile,pcx_struct *pcx)

// Decompress bitmap and store in buffer

{
  // Symbolic constants for encoding modes, with
  //  BYTEMODE representing uncompressed byte mode
  //  and RUNMODE representing run-length encoding mode:
  const int BYTEMODE=0, RUNMODE=1;

  // Buffer for data read from disk:
  const int BUFLEN=5*1024;
  int mode=BYTEMODE;  // Current encoding mode being used,
                      //  initially set to BYTEMODE
  int readlen;  // Number of characters read from file
  static unsigned char outbyte; // Next byte for buffer
  static unsigned char bytecount; // Counter for runs
  static unsigned char buffer[BUFLEN]; // Disk read buffer

// Calculate size of image:
int image_width=pcx->header.xmax - pcx->header.xmin + 1;
int image_height=pcx->header.ymax - pcx->header.ymin + 1;
long image_size=(long)image_width * image_height;

// Allocate memory for bitmap buffer and return -1 if
//  an error occurs:
if ((pcx->image=new unsigned char[image_size])==NULL)
  return(-1);
int bufptr=0; // Point to start of buffer
readlen=0;    // Zero characters read so far

// Create pointer to start of image:
unsigned char *image_ptr=pcx->image;

// Loop for entire length of bitmap buffer:
  for (long i=0; i<image_size; i++) {
    if (mode==BYTEMODE) {    // If we're in individual byte
                             //  mode....
      if (bufptr>=readlen) { // If past end of buffer...
        bufptr=0;            // Point to start

        // Read more bytes from file into buffer;
        // if no more bytes left, break out of loop
        if ((readlen=read(pcxfile,buffer,BUFLEN))==0)
          break;
      }
      outbyte=buffer[bufptr++]; // Next byte of bitmap
      if (outbyte>0xbf) {       // If run-length flag...
```

```
        // Calculate number of bytes in run:
        bytecount = (int)((int)outbyte & 0x3f);
        if (bufptr>=readlen) {  // If past end of buffer
          bufptr=0;             // Point to start

          // Read more bytes from file into buffer;
          //  if no more bytes left, break out of loop
          if ((readlen=read(pcxfile,buffer,BUFLEN))==0)
            break;
        }
        outbyte=buffer[bufptr++]; // Next byte of bitmap

        // Switch to run-length mode:
        if (--bytecount > 0) mode = RUNMODE;
      }
    }

    // Switch to individual byte mode:
    else if (--bytecount == 0) mode=BYTEMODE;

    // Add next byte to bitmap buffer:
    *image_ptr++=outbyte;
  }
return (0);
}
```

The load_palette() Function

The *load_palette()* function is much simpler. In fact, it's pretty much identical to the code we used for loading the color map from a Targa file, though we've put it in a separate function for consistency with the *load_image()* function. The text of the *load_palette()* function is in Listing 4-8.



Listing 4-8 The load_palette() function

```
void load_palette(int pcxfile,pcx_struct *pcx)
{
  lseek(pcxfile,-768L,SEEK_END);
  read(pcxfile,pcx->palette,3*256);
  for (int i=0; i<256; i++)
    for (int j=0; j<3; j++)
      pcx->palette[i*3+j]=pcx->palette[i*3+j]>>2;
}
```

There are two major differences between the palette loading code for the PCX files and that for the Targa files. Before we can load the PCX palette, we must move the DOS file pointer to a position 768 bytes from the end of the file, since the PCX palette isn't guaranteed to follow immediately after the bitmap.

(Generally, there is a single 0 byte between the bitmap and the palette, and we could probably get away with simply skipping this byte. However, to play it safe, we position the file pointer relative to the end of the file, a technique that should work with all 256-color PCX files.) The other difference is that the 3 bytes of the color descriptors are in the same blue, green, red order required by the VGA 256-color palette setting function, so we don't need to step through them backwards as we did when loading the Targa color map.

The PCXSHOW Program

As with our Targa-loading function, we'll demonstrate the PCX loader by writing a short program to display 256-color PCX images on the mode 13h display. This program, like the Targa display program, will only work with 256-color images of 320 by 200 or lower resolution. The text of the program PCXSHOW.CPP appears in Listing 4-9. Because it's essentially identical to the TGASHOW program described earlier, no detailed explanations of its inner workings are needed.

Listing 4-9 The PCXSHOW program

```
// PCXSHOW.CPP Version 1.1
// Display a PCX image on the mode 13h screen
//
// Written by Christopher Lampton
//   for GARDENS OF IMAGINATION (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  "screen.h"
#include  "pcx.h"

pcx_struct pcx;

void main(int argc,char *argv[])
{
  if (argc!=2) { // Are there 2 arguments on command line?

    puts("Wrong number of arguments.\n");
    exit(-1);

  }

  // Create pointer to video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);
```

```
  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Put display in mode 13h:
  setmode(0x13);

  // Clear display:
  cls(screen);

    if (loadPCX(argv[1],&pcx)) {
      setmode(oldmode);
      exit(1);
    }
  int image_width=pcx.header.xmax - pcx.header.xmin + 1;
  int image_height=pcx.header.ymax - pcx.header.ymin + 1;
  long image_size=(long)image_width * image_height;
  setpalette(pcx.palette);
  for (int y=0; y<image_height; y++) {
    for (int x=0; x<image_width; x++) {
      screen[y*320+x]=pcx.image[y*image_width+x];
    }
  }

  while (!kbhit());

  // Release memory
  delete pcx.image;

  // Restore old video mode:
  setmode(oldmode);
}
```

This program is available on the included disk as PCXSHOW.EXE.

Building a Bitmapped Maze

Now that we have the ability to store predesigned bitmaps on the disk and load them into the computer's memory, we can use this capability to construct fully bitmapped three-dimensional images of the interior of a maze on the mode 13h display. This is not a trivial task, however. Before we can accomplish it, we'll need to solve several technical problems that we ignored in creating the wireframe maze code in the previous two chapters.

One problem we rather obviously ignored was realism — or, more importantly, the lack thereof. To give you an idea of the sort of realism we're shooting for in our bitmapped maze engine, look at the screen shot in Figure 4-2, taken from a recent entry in the Wizardry series of CRPGs. This image is far more realistic than the wireframe graphics in the first five Wizardry games, which

we emulated with our wireframe maze program. Or go back and look at Figure 4-1, the one we used to demonstrate the TGASHOW program, which shows you some of the bitmaps that we'll be using.

Even better, go back into the BITMAP directory and use TGASHOW — or any other graphics display utility you own that can display 8-bit Targa files — to take another look at TESTMAZE.TGA. Quite a difference from the graphics in the last chapter, right? Instead of black lines on a white background, we now see fully detailed walls, a reflective tile floor, and a blue sky with puffy white clouds floating in it. In fact, this image is so detailed that you might wonder how we're going to cram all of those pixels into place fast enough to allow the user to walk around "inside" the maze in real time. The solution is to draw all of the pieces of the image in advance, then mix and match those pieces while the program is running to construct a picture of the way the maze appears from whatever location and direction the viewer is looking at it from.

And to do that, we'll turn to the Persistence of Vision ray tracer for help.

Ray Tracing the Maze

I've chosen to use the POV ray tracer to generate the maze images in this chapter for two reasons. The first is that it's a very good ray tracer. The second is that it's free.

If you're not already familiar with POV, that second statement may catch you off guard. Free? Who gives away perfectly decent pieces of software — especially software as complex and useful as a ray tracer — for free? In this case, the

Figure 4-2 A scene from Crusaders of the Dark Savant, the seventh entry in the Wizardry series. The bitmapped maze graphics are a considerable improvement over the wireframe graphics in the first five games of the series

samaritans who are giving away the software are a group of programmers who make their home on the CompuServe Information Service and who write graphics-related code for the sheer fun of it, not to mention the joy of providing useful, free software to people who have a need for it (or who just want to play around with it). Because it's available for free, you might want to get a copy of your own and use it to generate additional graphics for the maze program in this chapter (or for maze programs that you've written yourself). Take a look in Appendix A of this book for more detailed instructions on how to get your hands on POVRAY and how to use it to generate maze graphics for the program that we'll be creating in this chapter.

In case you're not already familiar with the concept, a ray tracer is a program that takes a mathematical description of a scene — a description couched in terms of certain standard shapes and the mathematical relationships between them — and creates a picture of that scene as it would appear if it existed in the real world. (Actually, the images created by a ray tracer often seem *more* realistic than any you might see in the real world, filled with impossibly shiny surfaces and vivid textures. This makes ray tracing an excellent method of generating hyper-realistic computer game images.) It does this by tracing the paths of imaginary light rays as they bounce around the scene, eventually encountering the viewer's eyes. We'll talk more about the mechanics of ray tracing in chapter 8. For now, we'll just use POVRAY as a turnkey application that can generate ray-traced images for us.

We haven't included POVRAY on the disk that comes with this book, and you don't really need a copy of it in order to use the maze engine that we'll develop in this chapter, because I've already created a set of images that are included on the disk. However, you may want to obtain a copy anyway, so that you can use it to generate new maze graphics for the program in this chapter (or for maze programs that you write yourself). Once again, you should take a look at Appendix A for details on how to do this.

Nine Views of the Maze

What sort of images do we want POVRAY to create? Well, the maze images that we're going to construct (as you can see in the TESTMAZE.TGA image) will consist of four primary elements: the floor, the sky, and the front and side walls of the "blocks" that make up the maze. Thus the images that we create with POVRAY will need to contain these four elements. The sky and floor will probably look pretty much the same in every image we construct of the interior of the maze, differing only in the way that they are obscured by the other two elements. It is the front and side views of the maze blocks that will differ most significantly from one image to the next. So the main thing we need POVRAY to

do for us is to generate pictures of the fronts and sides of the maze blocks in all the possible ways in which they may appear in a scene.

This may sound quite difficult, but in fact it only requires that we generate nine separate images. This will allow us to construct images of the maze as it appears out to five squares in front of the viewer, counting the square in which the viewer is standing. (We can actually get by with even fewer images than this,

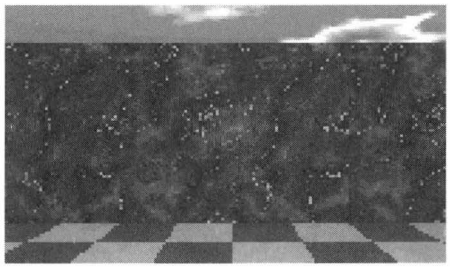


Figure 4-3a–e These five images, generated by the POV ray tracer, depict the fronts of maze blocks as they appear out to a distance of five squares in front of the viewer

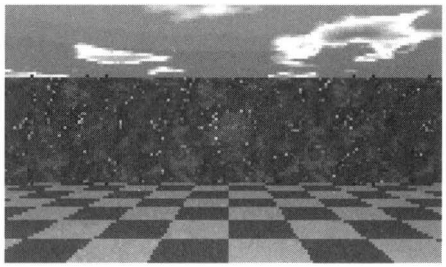


Figure 4-3b

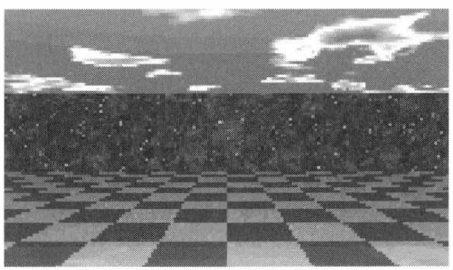


Figure 4-3c

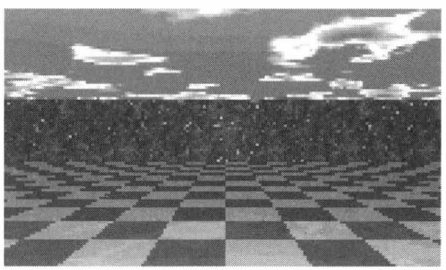


Figure 4-3d

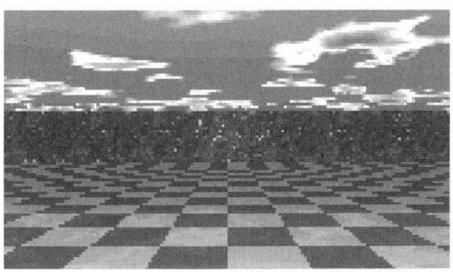


Figure 4-3e

as we'll explain at the end of this chapter, but for now we're going to tolerate some redundancy in the images in order to simplify our programming task.)

The first five images that we'll generate with POVRAY will depict the fronts of the maze blocks as they appear at distances of one, two, three, four, and five squares away from the viewer. These images appear in Figure 4-3. On the disk, we have given them the filenames FRONT1.TGA through FRONT5.TGA. You can view them using TGASHOW or any other file viewer that can display 8-bit Targa files.

The remaining four images depict the left and right sides of the maze blocks as they appear at distances of one, two, three, and four squares to each side of the viewer. It's not necessary to include images of the block sides as they appear at a distance of five squares from the viewer, since they'll no longer be visible on the periphery of the image at that distance. These images appear in Figure 4-4 and on disk as SIDE1.TGA through SIDE4.TGA.

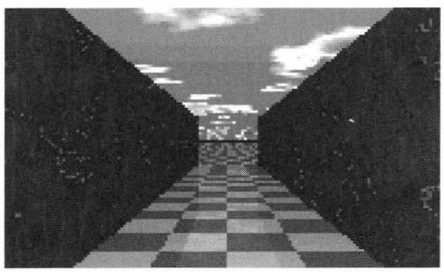


Figure 4-4a–d These four images depict the left and right sides of maze blocks as they appear at distances of one, two, three, and four squares to each side of the viewer

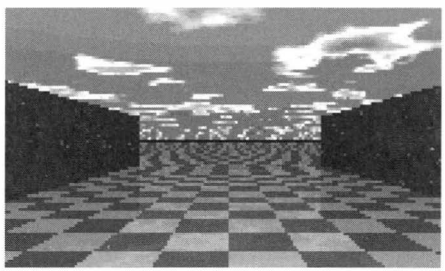


Figure 4-4b

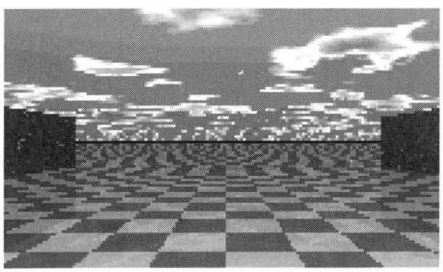


Figure 4-4c

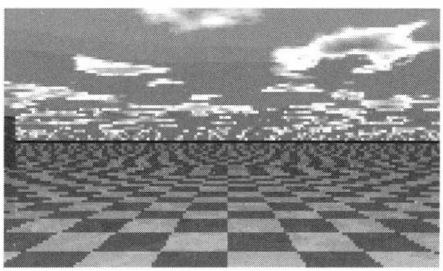


Figure 4-4d

Slicing and Dicing the Maze

Now that we have these images, what do we do with them? We chop them up and use them to construct individual "snapshots" of the maze. I like to think of this as the slice-and-dice method of maze generation. From these nine ray-traced images, we can slice and dice together a large number of composite images showing how different parts of the maze look when viewed from various angles. The trick is figuring out what parts of these images to slice and when to slice them.

To that end, we'll create a *drawmaze()* function that will operate essentially like the *drawmaze()* function that generated our wireframe maze. In fact, it will be possible to drop this function into a program nearly identical to the animated wireframe maze program in the last chapter. However, this function will be a great deal more complex than the wireframe maze-drawing function, not just because it will deal with bitmaps rather than lines — in fact, bitmaps are actually somewhat easier to put on the display than straight lines — but because we'll use it to render more complex views of the maze.

For instance, the wireframe maze-drawing function developed in chapter 2 had difficulty with any parts of the maze that were not along the line of squares proceeding away directly in front of the viewer. And it couldn't handle situations in which four empty squares were arranged together in a block. It could draw maze images analogous to the bitmapped image in Figure 4-5, but not like the ones in Figure 4-6. We could get away with that sort of approximation of reality in a wireframe maze program because wireframe mazes are inherently unrealistic. If holes appeared in the image where hallways were supposed to be, it wasn't especially bothersome. In a bitmapped maze program, however, we can't tolerate such lapses of realism.

To get around these problems, we'll use a recursive maze-drawing function. What's a recursive function? It's a function that calls itself. If you're not familiar with the principles of recursive programming, this might sound rather pointless. Why would we write a function that calls itself? As I describe the algorithm behind the function, though, I suspect you'll begin to see why recursion is just the ticket for producing a realistic maze.

The Maze-Drawing Algorithm

The maze-drawing function will be based on the simple concept that the three-dimensional view of the maze, which we'll be drawing in a rectangular window on the display, divides up neatly into three portions. Let's take a closer look at how it does so.

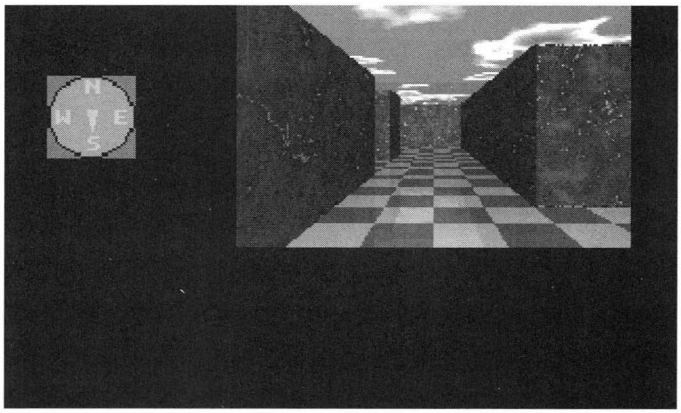
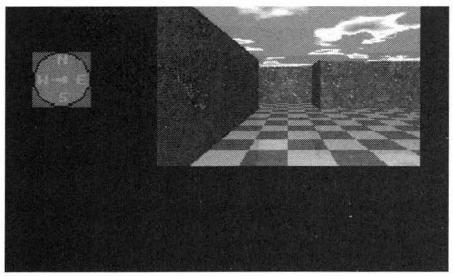


Figure 4-5 We could draw this image using essentially the same techniques we used to create the wireframe maze graphics in chapters 2 and 3, if we substituted bitmaps for white lines

As in the last two chapters, the maze will be defined as a matrix of squares similar to a checkerboard. Each of these squares is either empty or contains a solid block, presumably carved from some impassable (and visually opaque) substance. At any given moment, the viewer will be standing in the middle of one of these squares, looking in one of the cardinal compass directions — that is, toward one of the sides of the current square. We can therefore visualize the three-dimensional maze view that we'll be drawing on the screen as consisting of



Figures 4-6 a–b The techniques we used to create the wireframe maze graphics in chapters 2 and 3 would not be adequate to create images like these, even with the substitution of bitmaps for white lines

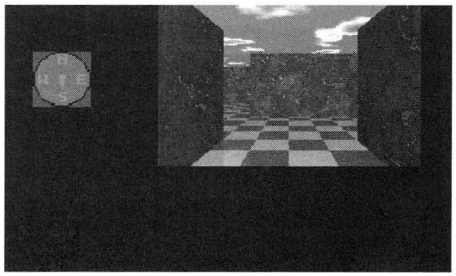


Figure 4-6b

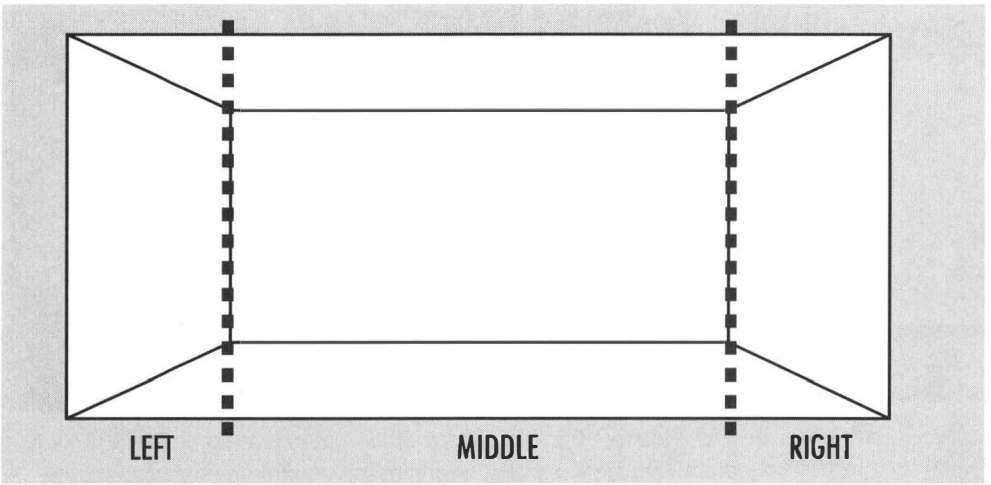


Figure 4-7 The maze view window divides neatly into left, right, and middle views

three parts: the view to the left of the current square, the view to the right, and the view straight down the middle. Figure 4-7 shows how the view window on the display divides up into these three views.

Each of these three views looks across one of the three visible squares adjoining the current square. (There are actually four squares adjoining the current square, but the one to the rear of the viewer isn't visible, so we won't be concerned with it here.) If one of these squares contains a block, then the portion of the view window that looks across that square will be completely filled with the image of that block, or a section of that image. If the square to the left of the current square contains a block, for instance, the left portion of the view will be filled with an image of the right side of that block (see Figure 4-8). If the square to the right contains a block, the right portion of the view will be filled with an image of the left side of that block (see Figure 4-9). And if the square in front of the current square contains a block, the middle portion of the view will be filled with an image of the front of that block (see Figure 4-10). Each of these images — of the left, right, and front sides of a maze block as seen at a distance of one square — are available in one of the TGA files that we created with the POV ray tracer. In this case the left and right images are in SIDE1.TGA and the front image is in FRONT1.TGA.

If all three of these squares contain blocks, then drawing the view will be a simple matter of slicing these images from the SIDE1.TGA and FRONT1.TGA files and dicing them into the appropriate portions of the view window. Figure 4-11 shows what the resulting image would look like. Drawing this image is simply a matter of checking the squares to the left, right, and in front of the

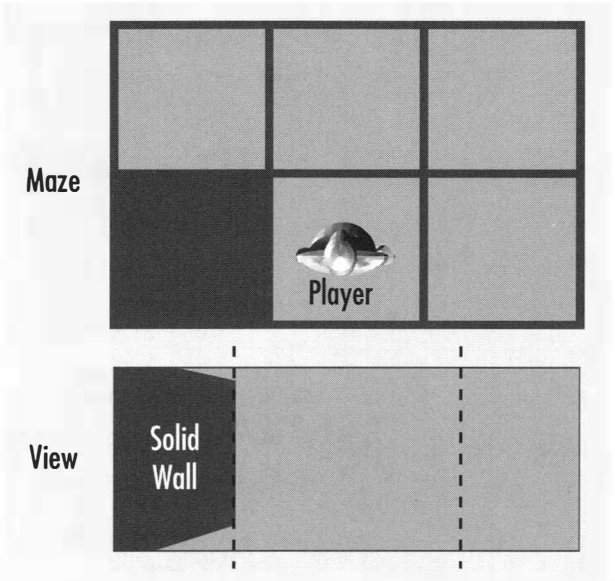


Figure 4-8 If the square to the left of the current square contains a block, the left portion of the view will be filled with an image of the right side of that block

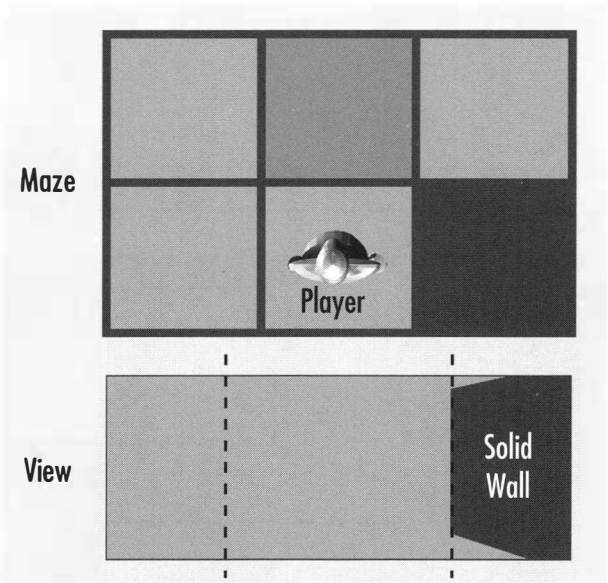


Figure 4-9 If the square to the right of the current square contains a block, the right portion of the view will be filled with an image of the left side of that block

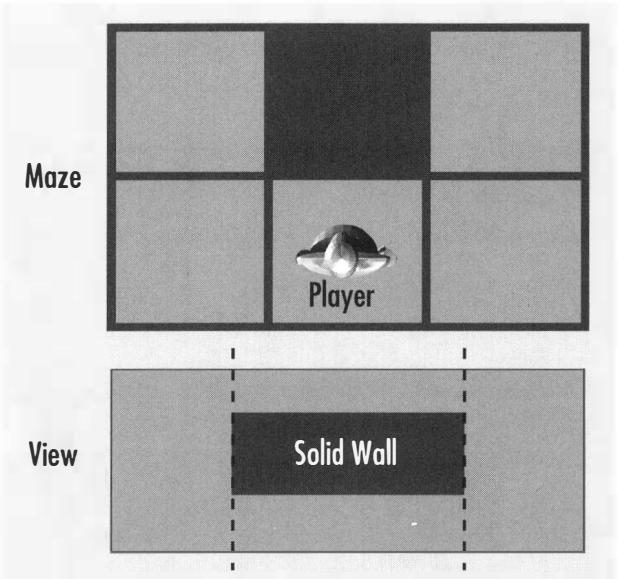


Figure 4-10 If the square in front of the current square contains a block, the middle portion of the view will be filled with an image of the front of that block

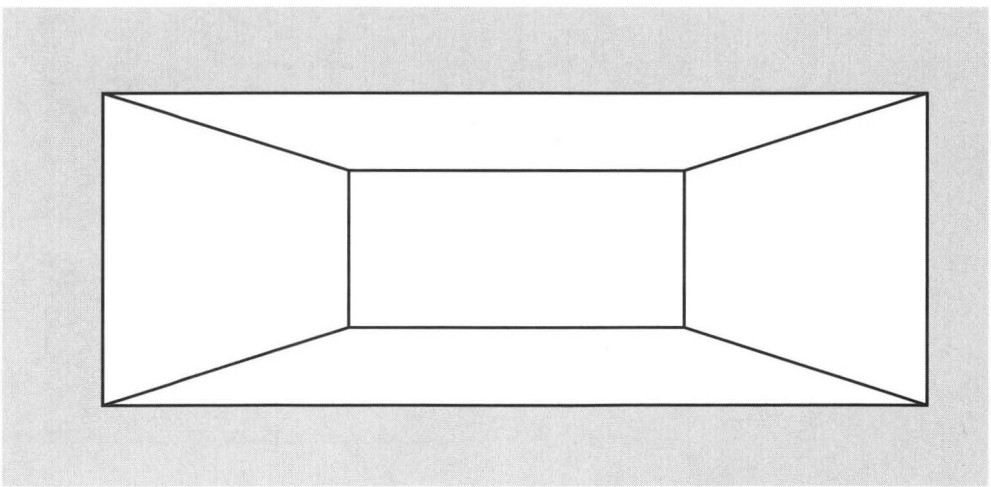


Figure 4-11 The view from the current square if all three visible adjoining squares contain blocks

current square to see if they contain blocks. If they do, the images of the blocks are copied from the TGA files (which we will have stored in memory previously, using the *loadTGA()* function) into the appropriate positions on the display.

But this is an unusually easy case to handle. In fact, it is the simplest image that our maze-drawing function will ever be called upon to render. What if one of the visible squares adjoining the current square is *not* occupied by a block? What do we do then? Obviously, we don't draw the image of a block if there isn't one there — but what do we draw in its place? We can't simply draw nothing; that would leave a gaping black hole in our drawing, which is inherently unrealistic. We might have been able to get away with this in our wireframe maze program, but such a lazy programming technique isn't adequate to the realism of a bitmapped maze.

We could simply proceed to check to see if the squares on the other side of the adjoining square are occupied, then draw any blocks that we find there, but this is an awkward thing to do. There may be as many as three additional squares visible through each of the squares adjoining the current square and we'll need to search each of them to see if they're occupied by blocks. And if they *aren't* occupied by blocks, we'll need to search even further into the maze, looking for blocks to draw.

If we're not careful, our *drawmaze()* function could become quite complicated. Fortunately, there's a fairly simple approach that we can take that will strip away much of the complexity. Each of the squares adjoining the current maze square presents almost exactly the same visual problem that the current square does — that is, each is bordered by three squares that could potentially contain visible blocks. (Once again, we'll ignore the fourth adjoining square, the one to the rear of the square currently under consideration. Although in many cases this square may be visible, the fact that we can see through it tells us that it isn't occupied by a block.) If we write a simple function that examines the current square (which we know to be empty because we're standing in it) to see if it's surrounded by any visible blocks (and which draws them if it finds them), that function can call itself recursively with slightly modified parameters to examine any adjoining empty squares to see if they are in turn surrounded by visible blocks. In other words, all we need to do is write a function to deal with a single square and let recursion handle the rest.

This will present some technical problems, but they aren't difficult to solve. We'll have to be sure that the function only draws those blocks that would be visible from the current position and that it doesn't try to draw blocks that are off the left and right edges of the view window. However, this is a simple matter of passing a pair of parameters to the function telling it what the left and right limits of the current view actually are. Initially, these will be the left and

right horizontal coordinates of the entire view window. (We'll measure these coordinates relative to the 160 by 100 pixel view window itself, so the left horizontal coordinate will initially be 0 and the right horizontal coordinate will be 159.) But when the function calls itself recursively, these will become the left and right limits of the part of the view window covered by the square currently being examined. When the function calls itself to handle the square to the immediate left, for instance, it will pass itself the left and right coordinates of the portion of the view window that represents the left view, as we illustrated in Figure 4-8.

The Bitmapped drawmaze() Function

Now that we have at least *some* idea of how the maze-drawing function will work, let's write it. The prototype for the function looks like this:

```
void drawmaze(int xshift,int yshift,int lview,int rview,
                  char *screen);
```

The first thing you're likely to notice is that there are a lot more parameters being passed to this version of the function than there were to the version that drew wireframe mazes. Bear in mind that this function is designed to work with only one square of the maze at a time; it will call itself recursively to deal with other squares in the maze. Thus the first two parameters, *xshift* and *yshift,* tell us how far the square currently being examined is from the square in which the viewer is standing. The first parameter, *xshift,* tells us how far (in squares) this square is to the left of the viewer's square (with negative values representing positions to the right of the viewer's square). The second parameter, *yshift,* tells us how far forward the square currently being examined is from the viewer's square. This value will always be positive, since we aren't interested in the unseen squares to the rear of the viewer's square.

On the first call to the function, these values should both be 0, since the square in which the viewer is standing is always the first square that *drawmaze()* should examine. During recursive calls, the function will shift these parameters to the left, right, and front, if the squares in those directions are visible and no blocks are found in them. The *lview* and *rview* parameters represent the left and right horizontal coordinates of the screen window in which any visible blocks are to be drawn. Initially, these will be the same as the left and right horizontal coordinates of the view window — that is, 0 and 159 — but during recursive calls they will be repeatedly reduced to those sections of the view window that are visible through the square currently being examined.

We don't want the function to keep calling itself recursively forever; this could cause it to lock up in an endless loop. As a more practical matter, we don't want it to call itself to examine squares beyond the visibility limit inherent in our

prerendered TGA files. Since these files only show the sides of blocks up to five squares distant, we'll set the viewing distance as covering the viewer's current square plus four additional squares. We'll define this distance at the front of the program with the constant VIEW_LIMIT (which we'll set equal to 4) and begin the *drawmaze()* function by checking to see if it has been reached:

```
// Return if we've reached the recursive limit:
if ((abs(xshift) > VIEW_LIMIT) || (abs(yshift) > VIEW_LIMIT))
   return;
```

This will prevent the function from making further recursive calls to itself. In practice, we don't want this limit ever to be reached, since invoking this escape clause will leave a gaping black hole in the maze drawing. So we must be careful to design mazes in which the viewer can never see more than four squares ahead in any direction. Later in this chapter, we'll suggest some ways to avoid — or at least make less restrictive — this limitation. (Note that this doesn't mean avoiding more than four squares in a row, but avoiding lines of sight that extend for more than four squares. Since lines of sight can go at an angle to the grid of squares, there will be situations where the corridor in front of the view extends for four or fewer squares, but a fifth square is nonetheless visible around a corner. It may require some trial and error to find such situations and restructure the map of the maze to avoid them.)

Now that we've got that minor technicality out of the way, we can get down to the real work of the function, which is to check the squares to the left, front, and right of the current square to see if they are occupied by blocks. We'll break this into three separate tasks and check the square to the left first. We'll do this with the aid of the same data structures that we used in the last two chapters to define positions and movements within the maze. You'll recall that the viewer's position is contained in the variables *pos.x* and *pos.y*, and that the viewer's orientation is contained in the variable *dir*. Because the meaning of such directions as left, right, and forward changes according to the viewer's orientation, we also created three arrays — called *left*, *right*, and *forward* — that defined the way in which the x and y coordinates within the maze change when we move in each of those directions. For instance, the variables *left[0].x* and *left[0].y* contain the changes in the x and y coordinates when our orientation is in direction 0 (which happens to be north) and we move to the left. Similarly, the variables *forward[2].x* and *forward[2].y* contain the changes in the x and y coordinates when our orientation is in direction 2 (south) and we move forward.

We can use these data structures to calculate coordinates within the maze of the square currently being examined and assign those coordinates to the variables *sx* and *sy*, like this:

```
// Calculate coordinates of square to left:
```

```
int sx = pos.x + (xshift+1) * left[dir].x + yshift * forward[dir].x;
int sy = pos.y + (xshift+1) * left[dir].y + yshift * forward[dir].y;
```

We multiply *xshift* by the left increment for the current position because *xshift* represents movement to the left of the viewer's position and we multiply *yshift* by the forward increment because *yshift* equals movement forward from the viewer's position (see Figure 4-12).

We know that the parameters *lview* and *rview* represent the left and right limits of our current drawing window. Now we must calculate the limits of the lefthand portion of that window, the portion that represents the view to the left of our current position. We'll store these limits in the integer variables *vleft* and *vright*. Calculating the left limit is easy, since it's the same as the left limit of our overall view window:

```
vleft=lview;
```

Calculating the right limit of the lefthand view is trickier — a *lot* trickier. In fact, there's not really any way to calculate it at all, since we're working with predefined bitmaps and we must match our view to the positions of the blocks within those bitmaps; otherwise, they won't be drawn properly when we slice and dice our images together. So, in order to see where the right edge of the lefthand view is, we must look at the images created by POVRAY to see where it drew the edges of the block.

If the square that we're currently examining is the one in which the viewer is standing, then the left portion of the view extends from horizontal coordinate 0

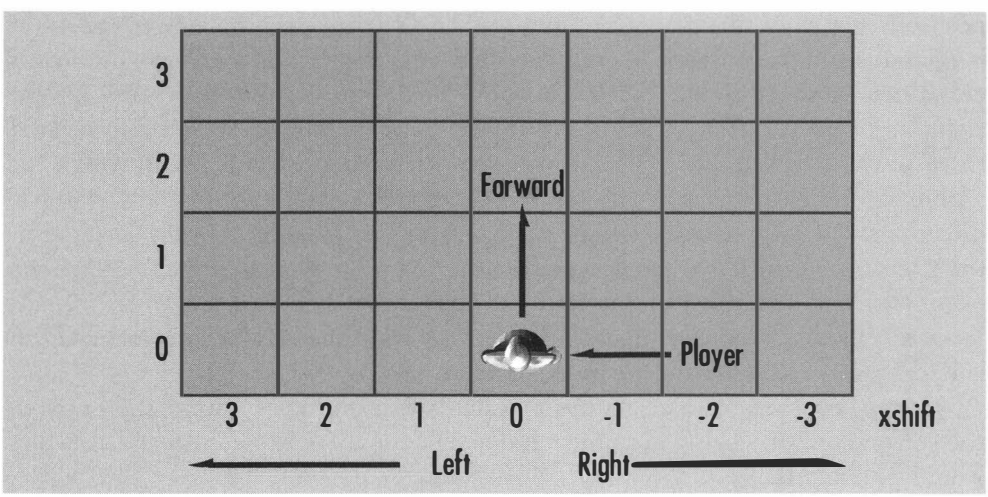


Figure 4-12 Moving left, right, and forward from the viewer's position

(the extreme left side of the view window) to horizontal coordinate 39. (These coordinates are defined relative to the contents of the window, not the display in which the window will eventually be placed.) You can see this by looking at FRONT1.TGA. Although it's hard to tell where one block ends and another begins by looking at the greenish granite walls of the maze, it's possible to identify block boundaries by looking at the checkerboard patterns on the floor. Each maze square contains 16 checkers, arranged four on a side. The four checkers aligned with the middle portion of the green wall define the block directly in front of the viewer. The block portions to the left and right of this define the positions of the two adjoining squares. If you count the pixels from the left edge of the view to the right edge of the leftmost block, you'll find that they extend from horizontal coordinate 0 to horizontal coordinate 39, as in Figure 4-13. (Well, okay, you're probably not enthusiastic about sitting around counting pixels on the display, so I've counted them for you using Deluxe Paint.) Similarly, the central portion of the view (corresponding to the extent of the block immediately in front of the viewer) extends from horizontal coordinate 40 to horizontal coordinate 120 and the righthand portion of the view extends from horizontal coordinate 121 to horizontal coordinate 159.

But these coordinates are only valid if we're examining the square where the viewer is standing. As the maze-drawing function moves recursively from square to square, the precise horizontal coordinates represented by the left, right, and center views will change. For instance, from the square to the immediate left of

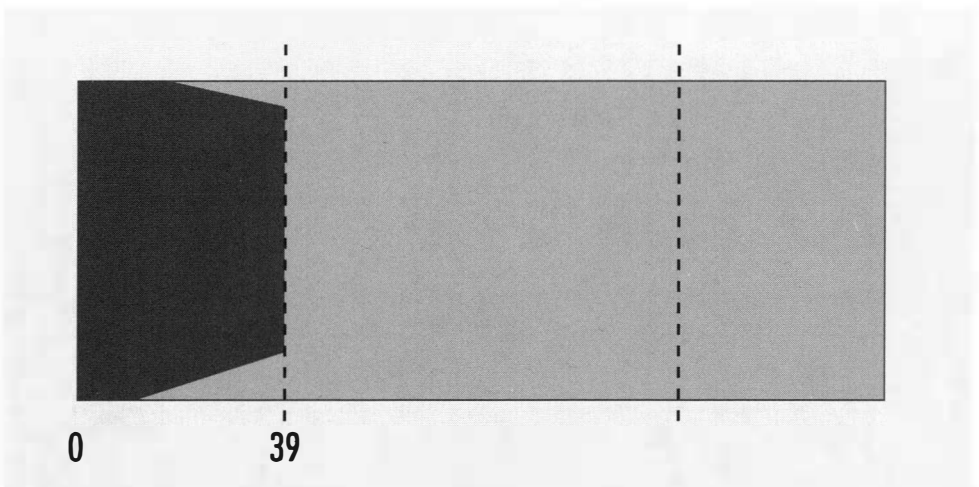


Figure 4-13 The leftmost block in FRONT1.TGA extends across the image from horizontal position 0 to horizontal position 39

the current square, the entire view covers only the lefthand portion of the display — that is, the area from horizontal coordinate 0 to horizontal coordinate 39. Within that portion, the lefthand portion of the view from that square only covers the segment of the view window from horizontal coordinate 0 to horizontal coordinate 7. (This is the extent of the leftmost block front in FRONT2.TGA.) The central portion of the view extends from horizontal coordinate 8 to horizontal coordinate 39. And there *is* no righthand portion to the view from that square, since it would be outside the area that we can view *through* that square.

Whew! How are we going to take care of all that in our recursive function? How can we determine where the lefthand, righthand, and central portions of the current view are in the view window? The answer is surprisingly simple. We must treat the grid of maze squares radiating outward in front of the view as a network of intersecting grid lines, as shown in Figure 4-14. All we need to worry about are the screen coordinates of the points at which these grid lines intersect, since these correspond to the segments of the view window taken up by the fronts and sides of the maze blocks. We can determine these by studying the TGA files produced by the ray tracer using a program like Deluxe Paint. Then we can encode them in a two-dimensional array that can be referenced by the *drawmaze()* function. Don't worry; you won't have to dig up a copy of Deluxe Paint and do this yourself, because I've already done it for you. Here's the resulting array:

```
int matrix[VIEW_LIMIT+1][VIEW_LIMIT*2+2]={
```

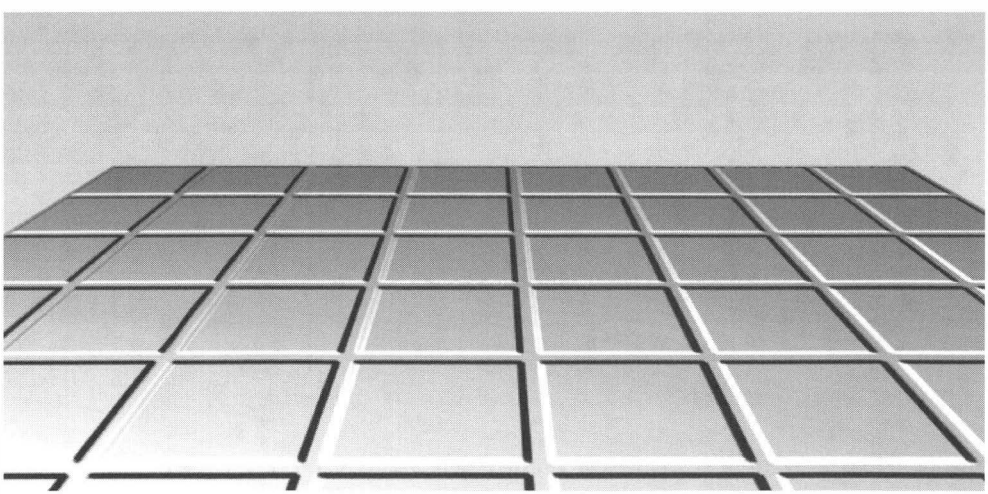


Figure 4-14 The network of grid lines extending out in front of the viewer

```
  {160,160,160,160,121, 40,  0,  0,  0,  0},
  {160,160,160,153,105, 56,  8,  0,  0,  0},
  {160,160,160,132, 98, 63, 29,  0,  0,  0},
  {160,160,147,121, 94, 67, 40, 14,  0,  0},
  {160,160,141,117, 93, 68, 44, 20,  0,  0}
};
```

The first row of elements in this array corresponds to the grid line running horizontally in front of the viewer — that is, to the row of block fronts shown in the image FRONT1.TGA. There are only two visible intersection points between this grid line and the vertical grid lines running away from the viewer. These occur at horizontal coordinates 40 and 121 in the view window. We've placed those numbers in the center of the row and assigned the other intersection points horizontal coordinates corresponding to the edge of the display, indicating that they are actually *off* of the edge. The second row of elements corresponds to the horizontal row behind that row, equivalent to the row of block fronts shown in the image FRONT2.TGA. And so forth. The two middle elements of each array correspond to the intersection points between these rows and the rows of block sides depicted in SIDE1.TGA. The two elements to each side of these correspond to the intersection points with the rows of block sides in SIDE2.TGA. And so forth.

This gives us a relatively simple way of determining the current left portion of the display. We can do it like this:

```
vright=matrix[yshift][xshift+VIEW_LIMIT+1];
```

This gives us the position in the *matrix[]* array corresponding to the grid intersection immediately forward and to the left of the square currently being examined.

It's possible that the right edge that we've calculated in this manner will actually be farther to the right than the right edge of the viewing window passed to us by the calling function. If so, the edge passed by the calling function takes priority over this edge; we must clip this edge to it:

```
if (vright>rview) vright=rview;
```

After all of the above, the width of our new viewing window may be 0; this would be true, for instance, if we've recursed all the way past the left edge of the original view window, into that part of the grid where we've set all intersection points equal to either 0 or 160. So we calculate the width of the window:

```
vwidth=vright-vleft;
```

If the width isn't 0, we proceed to check the square to the left of the current square — the one that's visible through the left portion of the current view — to see if it's occupied by a block:

```
if (vwidth>0) {
  if (maze[sx][sy]) {
```

If it is, we call a function called *display_slice()* that takes a horizontal slice out of one of the TGA images and copies it to the video buffer:

```
  display_slice(xshift+VIEW_LIMIT+1,vleft,vright,screen);
}
```

We'll look at the *display_slice()* function in a moment. The first parameter in the call is the number of the file containing the desired slice, where FRONT1.TGA through FRONT5.TGA are 0 through 4 and SIDE1.TGA through SIDE4.TGA are 5 through 8. We're drawing the block to the left of the current block, so we know that we'll be drawing a side view. Thus the file that we want to slice the image from is at least number 5, which is VIEW_LIMIT+1. (We use VIEW_LIMIT+1 to determine the number of the file rather than 5 so that it will be easier to change the view limit at a later date, though we'll need to add additional bitmap files should we decide to do this.) We add to this the *xshift* distance from the viewer's position, since this will define how many squares away from the center the side that we wish to draw is and thus which TGA file it will come from. The remaining three parameters define the left and right horizontal coordinates of the slice that we wish to draw and the address of the screen buffer we wish to draw it to.

What if we don't find a block in the square to the left? Simple. We call the *drawmaze()* function recursively, shifting one square to the left, and start all over again:

```
  else drawmaze(xshift+1,yshift,vleft,vright,screen);
}
```

Now we just repeat this process for the central and rightmost portions of the view. Here's how we handle the center portion of the view:

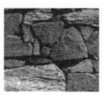
```
// Get left-right extent of center part of view:
vleft=matrix[yshift][xshift+VIEW_LIMIT+1];
vright=matrix[yshift][xshift+VIEW_LIMIT];
if (vleft<lview) vleft=lview;
if (vright>rview) vright=rview;
vwidth=vright-vleft;

if (vwidth>0) {

  // If the square directly ahead is occupied,
  // draw front of cube:
  if (maze[sx][sy]) {
    display_slice(yshift,vleft,vright,screen);
  }
```

```
  // Else call function recursively to draw the view from
  // next square forward:
  else drawmaze(xshift,yshift+1,vleft,vright,screen);
}
```

And we handle the rightmost portion of the view in a nearly identical manner:

```
  // Calculate coordinates of square to right:
  sx = pos.x + (xshift-1) * left[dir].x + yshift * forward[dir].x;
  sy = pos.y + (xshift-1) * left[dir].y + yshift * forward[dir].y;

  // Get left-right extent of right part of view:
  vleft=matrix[yshift][xshift+VIEW_LIMIT];
  if (vleft<lview) vleft=lview;
  vright=rview;
  vwidth=vright-vleft;

  if (vwidth>0) {

    // If the square to the right is occupied,
    // draw left side of cube:
    if (maze[sx][sy]) {
      display_slice(VIEW_LIMIT+1-xshift,vleft,vright,screen);
    }

    // Else call function recursively to draw the view from
    // next square to the right:
    else drawmaze(xshift-1,yshift,vleft,vright,screen);
  }
}
```

The complete text of the *drawmaze()* function appears in Listing 4-10.



Listing 4-10 The bitmapped drawmaze() function

```
void drawmaze(int xshift,int yshift,int lview,int rview,char *screen)

// Recursive function to draw left, right and center walls for the
// maze square at POS.X+XSHIFT, POS.Y+YSHIFT, as viewed from POS.X,
// POS.Y, within the screen window bordered by horizontal coordinates
// LVIEW and RVIEW on the left and right respectively. When called
// with XSHIFT and YSHIFT set to 0, draws entire maze as viewed
// from POS.X,POS.Y out to VIEW_LIMIT squares.
{
  int vleft,vright,vwidth;

  // Return if we've reached the recursive limit:
  if ((abs(xshift) > VIEW_LIMIT) || (abs(yshift) > VIEW_LIMIT))
```

continued on next page

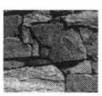
continued from previous page

```
    return;

  // Calculate coordinates of square to left:
  int sx = pos.x + (xshift+1) * left[dir].x + yshift * forward[dir].x;
  int sy = pos.y + (xshift+1) * left[dir].y + yshift * forward[dir].y;

  // Get left-right extent of left part of view:
  vleft=lview;
  vright=matrix[yshift][xshift+VIEW_LIMIT+1];
  if (vright>rview) vright=rview;
  vwidth=vright-vleft;

  if (vwidth>0) {

    // If the square to the left is occupied,
    // draw right side of cube:
    if (maze[sx][sy]) {
      display_slice(xshift+VIEW_LIMIT+1,vleft,vright,screen);
    }

    // Else call function recursively to draw the view from
    // next square to the left:
    else drawmaze(xshift+1,yshift,vleft,vright,screen);
  }

  // Calculate coordinates of square directly ahead:
  sx = pos.x + xshift * left[dir].x + (yshift+1) * forward[dir].x;
  sy = pos.y + xshift * left[dir].y + (yshift+1) * forward[dir].y;


  // Get left-right extent of center part of view:
  vleft=matrix[yshift][xshift+VIEW_LIMIT+1];
  vright=matrix[yshift][xshift+VIEW_LIMIT];
  if (vleft<lview) vleft=lview;
  if (vright>rview) vright=rview;
  vwidth=vright-vleft;

  if (vwidth>0) {

    // If the square directly ahead is occupied,
    // draw front of cube:
    if (maze[sx][sy]) {
      display_slice(yshift,vleft,vright,screen);
    }

    // Else call function recursively to draw the view from
    // next square forward:
    else drawmaze(xshift,yshift+1,vleft,vright,screen);
  }

  // Calculate coordinates of square to right:
  sx = pos.x + (xshift-1) * left[dir].x + yshift * forward[dir].x;
  sy = pos.y + (xshift-1) * left[dir].y + yshift * forward[dir].y;
```

```
  // Get left-right extent of right part of view:
  vleft=matrix[yshift][xshift+VIEW_LIMIT];
  if (vleft<lview) vleft=lview;
  vright=rview;
  vwidth=vright-vleft;

  if (vwidth>0) {

    // If the square to the right is occupied,
    // draw left side of cube:
    if (maze[sx][sy]) {
      display_slice(VIEW_LIMIT+1-xshift,vleft,vright,screen);
    }

    // Else call function recursively to draw the view from
    // next square to the right:
    else drawmaze(xshift-1,yshift,vleft,vright,screen);
  }
}
```

The draw_slice() Function

The text of the *draw_slice()* function, which draws a vertical slice out of a 160 by 100 bitmap stored in an array to the equivalent position in the 160 by 100 view window, appears in Listing 4-11.



Listing 4-11 The draw_slice() function

```
void display_slice(int imagenum,int leftx,int rightx,char far *screen)

// Displays vertical slice of 160x100 bitmap from horizontal
// coordinate LEFTX to horizontal coordinate RIGHTX
{
  for (int y=0; y<100; y++) {
    for (int x=leftx; x<(rightx+1); x++) {
      screen[(YWINDOW+y)*320+XWINDOW+x]=
        tga[imagenum].image[y*160+x];
    }
  }
}
```

The bitmaze() Program

We can construct a simple program along the lines of the GOMAZE program from the last chapter to allow a user to take a tour of the bitmapped maze that we

draw with the *drawmaze()* function. In fact, much of the code in the *main()* function of this program will be identical to that in the *main()* function of the GOMAZE program. However, there are some differences. For instance, the BITMAZE program will need to allocate an array of variables of type *tga_struct* to hold the Targa images that we must load from the disk:

```
// Array for bitmaps:
tga_struct tga[VIEW_LIMIT*2+1];
```

And we'll need to store the names of the nine TGA files in an array, so that we can load them using a *for()* loop:

```
// Array of names for bitmap files:
char *tga_name[VIEW_LIMIT*2+1]=
     {"front1.tga",
      "front2.tga",
      "front3.tga",
      "front4.tga",
      "front5.tga",
      "side1.tga",
      "side2.tga",
      "side3.tga",
      "side4.tga"
     };
```

We'll also need a couple of PCX structures. One, which we'll call *bg*, will hold the background image that you saw earlier in Figure 4-2. This image is in the BITMAP directory in the file MAZE.PCX and can be viewed with the PCXSHOW program. We'll also need a structure to hold the image in the file COMPASS.PCX. This image, which can also be viewed with PCXSHOW (and which appears in Figure 4-15), contains four compass faces, with the compass needle pointing in a different cardinal direction in each. In a moment, we'll move these compass faces into an array of compass faces. The PCX structures and array of compass faces will be declared like this:

```
// Buffers for compass picture and background:
pcx_struct bg,compasses;

// Array of compass faces:
unsigned char far *compass_face[4];
```

The compass images will be moved from the PCX file where they are stored into the elements of the *compass_face* array using a short function called *grab*, which appears in Listing 4-12. This short function, which we'll put in a new module devoted to bitmap-related functions called BITMAP.CPP, grabs a rectangular portion of a 320 by 200 mode 13h bitmap from a buffer and copies it into a buffer. The parameters are the upper-left coordinates of the rectangle to be

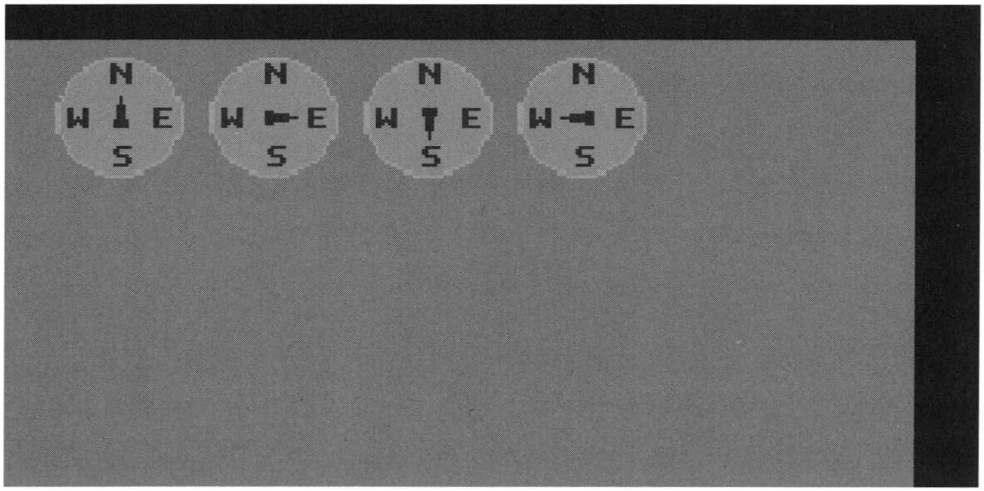
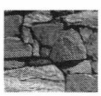


Figure 4-15 The bitmapped maze depicted by BITMAZE

grabbed, the width and height of the bitmap, a pointer to the buffer holding the 320 by 200 image, and a pointer to the buffer into which the bitmap is to be copied.



Listing 4-12 The grab() function

```
void grab(int x1,int y1,int width,int height,
          unsigned char far *screen,
          unsigned char far *buffer)
{
  for (int y=0; y<height; y++)
    for (int x=0; x<width; x++)
      buffer[y*width+x]=screen[(y1+y)*320+x1+x];
}
```

Once we've got the compass face images in the array, we'll move them onto the display using another function from our BITMAP.CPP module, called *blit()*. The word "blit" is short for "bit block transfer," a commonly used term for the process of moving rectangular bitmaps between computer memory and video memory, or from one location in video memory (or an offscreen video buffer) to another. The text of the *blit()* function, which is quite similar to the *grab()* function, appears in Listing 4-13.

Listing 4-13 The blit() function

```
void blit(int x1,int y1,int width,int height,unsigned char far *screen,
          unsigned char far *sprite)
{
  for (int y=0; y<height; y++)
    for (int x=0; x<width; x++) {
      unsigned char byte=sprite[y*width+x];
      if (byte) screen[(y1+y)*320+x1+x]=byte;
    }
}
```

The *blit()* function treats values of 0 in a bitmap as transparent — that is, it doesn't copy them to video memory. It does this by checking to see if *byte* is nonzero before it copies the pixel value. This allows us to create rectangular images with nondisplaying areas in them, through which the background can show.

We'll also need to create a new maze, since the one we used in the last two chapters doesn't fit our needs anymore. Most importantly, we need a maze expressly designed so that the viewer cannot see more than five squares in any direction. Here is an array containing such a maze:

```
// MAP OF THE MAZE
//
// Each element represents a physical square within the maze.
// A value of 0 means the square is empty; a nonzero value
// means that the square is filled with a solid block. Map
// must be designed so that no square can be viewed over a
// greater distance than that defined by VIEW_LIMIT.

unsigned char maze[16][16]={
     {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
     {1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,1},
     {1,0,1,0,0,0,1,0,0,0,1,0,1,1,0,1},
     {1,0,1,1,1,1,1,0,0,0,1,0,0,0,1,1},
     {1,0,0,0,0,1,0,0,0,0,0,1,0,1,0,1},
     {1,0,0,0,0,1,0,1,1,1,1,1,0,0,0,1},
     {1,1,1,0,0,1,1,0,1,0,0,1,1,0,1,1},
     {1,0,1,0,1,1,1,0,0,0,1,0,0,0,1,1},
     {1,0,1,0,0,0,1,1,1,1,0,0,0,1,1,1},
     {1,0,0,1,1,0,1,0,1,1,1,0,1,1,1,1},
     {1,1,0,0,0,0,1,0,1,0,0,0,0,0,1,1},
     {1,0,0,0,1,1,1,0,1,0,1,1,1,0,1,1},
     {1,0,1,1,1,0,1,0,0,0,0,1,0,0,0,1},
     {1,0,0,0,1,0,1,1,1,1,0,1,0,0,0,1},
     {1,1,0,0,0,1,1,0,0,0,0,1,0,0,0,1},
     {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
};
```

We'll place the viewer initially at a coordinate position of 3,7 within the maze:

```
// Viewer's starting position:
struct xy pos={3,7};
```

We'll also start out with the viewer facing north:

```
int dir=0;  // Viewer's current heading in maze,
            // where 0=north,1=east,2=south,3=west
```

As the *main()* function begins, we'll use the *loadPCX()* function to load in the file containing the compass face images and the *grab()* function to move those images into the *compass_face* array:

```
// Load compass picture:
if (loadPCX("compass.pcx",&compasses)) exit(1);

// Load compass faces into array:
for (int num=0; num<4; num++) {
  compass_face[num]=new unsigned char[42*41];
  grab(17+num*50,18,42,41,compasses.image,compass_face[num]);
}
```

We'll then use the *delete* function to get rid of the memory occupied by the COMPASS.PCX image, since it's no longer needed:

```
// Dispose of compass picture:
delete compasses.image;
```

We'll also use the *loadPCX()* function to load the background image into memory:

```
// Load background image:
if (loadPCX("maze.pcx",&bg)) exit(1);
```

And we'll use the *loadTGA()* function, along with the file names that we stored in the *tga_name* array, to load in the ray-traced images of the maze:

```
// Load maze bitmaps into TGA array:
for (unsigned int i=0; i<(VIEW_LIMIT*2+1); i++)
  if (loadTGA(tga_name[i],&tga[i])) exit(1);
```

Then we'll engage in exactly the same setup that we used in the GOMAZE program in the last chapter, except that we'll copy the background image into the offscreen video buffer before moving the image into video memory:

```
// Put background image into buffer:
  for (i=0; i<64000; i++)
    screen_buffer[i]=bg.image[i];
```

We'll draw an initial image before we begin the main *while()* loop that drives the game:

```
// Put screen buffer on screen:
blitscreen(screen_buffer);
```

The main *while()* loop is identical to the one in GOMAZE, except that we blit a compass face onto the display every time the loop executes:

```
// Put compass next to maze window:
blit(20,35,42,41,screen,compass_face[dir]);
```

And we call our new *drawmaze()* function with its new parameters:

```
// Draw the maze in screen buffer:
drawmaze(0,0,0,159,screen_buffer);
```

The complete text of the new *main()* function appears in Listing 4-14. The complete program is available on disk as BITMAZE.CPP in the BITMAP directory.

Listing 4-14 The main() function from BITMAZE.CPP

```
void main(void)
{
  event_struct events;
  struct xy newpos;

  // Create pointer to video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Create offscreen video buffer:
  char far *screen_buffer=new unsigned char [64000];

  // Load compass picture:
  if (loadPCX("compass.pcx",&compasses)) exit(1);

  // Load compass faces into array:
  for (int num=0; num<4; num++) {
    compass_face[num]=new unsigned char[42*41];
    grab(17+num*50,18,42,41,compasses.image,compass_face[num]);
  }

  // Dispose of compass picture:
  delete compasses.image;

  // Load background image:
  if (loadPCX("maze.pcx",&bg)) exit(1);

  // Load maze bitmaps into TGA array:
  for (unsigned int i=0; i<(VIEW_LIMIT*2+1); i++)
    if (loadTGA(tga_name[i],&tga[i])) exit(1);

  // Initialize event manager:
  init_events();
```

```
  // Calibrate the user's joystick:
  if (WHICH_EVENTS & JOYSTICK_EVENTS) {
    printf("\nCenter your joystick and press button ");
    printf("one.\n");
    setcenter(); // Calibrate the center position
    printf("Move your joystick to the upper lefthand ");
    printf("corner and press button one.\n");
    setmin();    // Calibrate the minimum position
    printf("Move your joystick to the lower righthand ");
    printf("corner and press button one.\n");
    setmax();    // Calibrate the maximum position
}


  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Put display in mode 13h:
  setmode(0x13);

  // Set palette to PCX palette:
  setpalette(tga[0].color_map);

  // Clear display:
  cls(screen_buffer);

  // Put background image into buffer:
  for (i=0; i<64000; i++)
    screen_buffer[i]=bg.image[i];

  // Set number of ticks per frame:
  ticks_per_frame = CLK_TCK / FRAMES_PER_SECOND;

  // Initialize the frame timer:
  clock_t lastframe=clock();

  // Make sure we get at least one frame
  // into the maze:
  events.quit_game=0;

  // Put initial image of maze in screen buffer:
  drawmaze(0,0,0,159,screen_buffer);

  // Put screen buffer on screen:
  blitscreen(screen_buffer);

  // Let's go for a walk in the maze:
  while(!events.quit_game) {

    // Draw the maze in screen buffer:
    drawmaze(0,0,0,159,screen_buffer);

    // Move screen buffer to screen:
    putwindow(XWINDOW,YWINDOW,160,100,screen_buffer);
```

continued on next page

continued from previous page

```
    // Put compass next to maze window:
    blit(20,35,42,41,screen,compass_face[dir]);

    // Pause until time for next frame:
    while ((clock()-lastframe)<ticks_per_frame) ;

    // Start timing another frame:
    lastframe=clock();

    // Move viewer according to input events:
    getevent(WHICH_EVENTS,&events);

    // Do we want to move forward?
    if (events.go_forward) {
      newpos.x=pos.x+forward[dir].x;
      newpos.y=pos.y+forward[dir].y;
      if (!maze[newpos.x][newpos.y]) {
        pos.x=newpos.x;
        pos.y=newpos.y;
      }
    }

    // ...or do we want to move backward?
    else if (events.go_back) {
      newpos.x=pos.x-forward[dir].x;
      newpos.y=pos.y-forward[dir].y;
      if (!maze[newpos.x][newpos.y]) {
        pos.x=newpos.x;
        pos.y=newpos.y;
      }
    }

    // Do we want to turn left?
    if (events.go_left) {
      --dir;
      if (dir<0) dir=3;
    }

    // ...or do we want to turn right?
    else if (events.go_right) {
      dir++;
      if (dir>3) dir=0;
    }
  }

  // Terminate event manager:
  end_events();

  // Release memory
  for (int ii=0; ii<(VIEW_LIMIT*2+1); ii++) {
    delete tga[ii].image;
    delete tga[ii].color_map;
```

```
    if (tga[ii].ID) delete tga[ii].ID;
}
delete bg.image;
for (int jj=0; jj<4; delete compass_face[jj++]);
delete screen_buffer;

  // Restore old video mode:
  setmode(oldmode);
}
```

Touring the Maze

The bitmapped maze program works exactly like the wireframe maze program we presented in the last chapter. Go to the BITMAP directory and type:

```
bitmaze
```

The bitmapped maze screen will appear (see Figure 4-16). Use the keyboard, the mouse, or the joystick to move around through the maze. (The keyboard is recommended.) When you get tired of running around through ray-traced passageways, hit ESC to return to the DOS prompt.

While you're in the maze, be sure to notice some of the highpoints of the tour. My own favorite part of the maze are the reflections on the polished floor, added courtesy of the POV ray tracer. Also notice that you can never see more than four squares in front of the one you're standing in. If you're curious what would happen if you could see further than this, try making random changes to the maze array to see what happens. This won't crash the program — at least, it *shouldn't,* since we look for this possibility at the beginning of the *drawmaze()* function — but it will produce some odd effects.

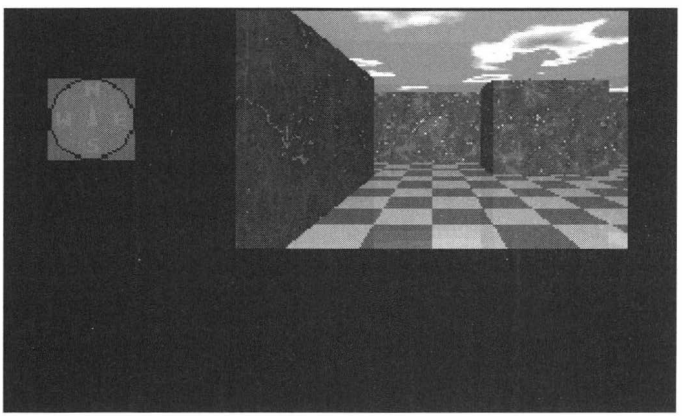


Figure 4-16 The bitmapped maze depicted by BITMAZE

Should you become bored with the green walls, checkered floors, and blue skies in this maze, I've supplied some files on the disk that you can use to produce new maze graphics — *if* you have the POV ray tracer. Full instructions are included in Appendix A for using POV to generate new mazes. In addition, I've provided batch files to automate the process, and a few alternate maze files to generate mazes with quite different appearances from the default maze supplied on the disk. These are also explained in Appendix A.

Refining the Maze Engine

Although we'll ring some interesting variations on it in later chapters, we won't be adding any more bells and whistles to this particular maze engine; instead, we're going to move on to more advanced types of maze code. Nonetheless, the code presented in this and the preceding two chapters could make the core of a pretty good commercial game engine, which could be used to produce maze games along the lines of Eye of the Beholder or Crusaders of the Dark Savant.

Still, this engine suffers from a few inefficiencies, some of which we remarked on earlier. These will need to be fixed if you want to use this code as a model for a maze engine of your own.

Perhaps the most important of these inefficiencies is in the way it uses memory. Storing nine complete 160 by 100 TGA images from which to construct the views of the maze is not a particularly efficient way to handle bitmaps. A quick glance will tell you that much of what appears in these bitmaps is redundant. The sky and floor, for instance, are repeated from image to image, when they really only need to be stored once. Of course, this allows us to pull off some nifty effects, like the reflections of the maze walls that appear in the shiny tiled floor. However, losing this reflection effect would be a small price to pay in order to use memory more efficiently. The best way to remove the floor and sky redundancy is to store an image of the floor and sky in a separate TGA (or PCX) file and move that image into the view window before drawing any of the walls, then draw the walls over the top of this image using the *blit()* function or an equivalent function that recognizes transparent pixels in a bitmap.

The wall images themselves are stored inefficiently. The SIDE4.TGA file, for instance, barely has any walls in it at all — and some of the other SIDE images have large portions without walls in them. A way around this inefficiency would be to load the TGA images in briefly during program initialization, slice the useful portions of these bitmaps out with the *grab()* function and store them in smaller arrays, then throw the rest of the images away. This is left, inevitably, as an exercise for the reader. The multiple FRONT images are also probably not necessary. One front-of-block image could be stored at each distance, then moved horizontally so

that it could stand in for all of the other front images at that distance. (In chapter 6, we'll look at a method of reducing the storage requirements even further by taking only a single front image and using that image to generate not only all of the other front images but the side images as well.)

Yet another way to save memory would be to compress the graphics data, to make the memory use even more efficient. You could use an RLE compression system similar to that used in PCX files. Decompressing graphics will slow the display functions down slightly, but high frame rates aren't as important in bitmapped maze programs like this one as they are in the flight simulator-style programs we'll be demonstrating later. Because the player moves through the maze by quantum-leaping from square to square, flashing the frames on the screen too quickly would cause the player to zip across the maze in only one or two seconds, which wouldn't be desirable.

Compressing the graphics data could in turn help you cure another problem with this maze engine — the inability to see more than four squares beyond the square in which you are standing. This is, in many ways, a memory limitation: We don't have room to store any more TGA images without overflowing the limited space that DOS provides us with in the first 640 kilobytes of the computer's memory. Each 160 by 100 TGA image takes up 16 kilobytes of memory. That may not sound like much, but it adds up. BITMAZE uses about 144 kilobytes of memory for TGA images, plus an additional 64 kilobytes for the background image loaded from the file MAZE.PCX. More than this we really don't want to spare for graphics. But if you can find a more efficient way to use memory — by throwing out the redundant parts of the TGA images, for instance, or by compressing the graphics — you could store images of the maze walls out to five, six, seven, or more squares in front of the viewer. (An alternate way to handle this problem is to design the maze graphics so that they seem to fade into darkness with distance. Then it would appear realistic that the viewer could only see four squares ahead. However, I was unable to tweak the POV ray tracer into providing this particular effect, so you may need to design such images by hand, using a paint program.)

Perhaps even more importantly, you will want to provide a greater variety of wall images, especially images of walls with doors in them, or with decorations mounted on them. This will provide a more interesting visual environment for the game player. These alternate images could be taken into account in the *maze[]* array by using numbers other than 1 to designate squares filled with blocks. For instance, the number 2 could indicate a block with a door in it, the number 3 could indicate a block with wall sconces on it, and so on. However, this will require storing still more bitmaps, which will put a tight squeeze on your lower 640 kilobytes of memory even with more efficient methods of storing graphics.

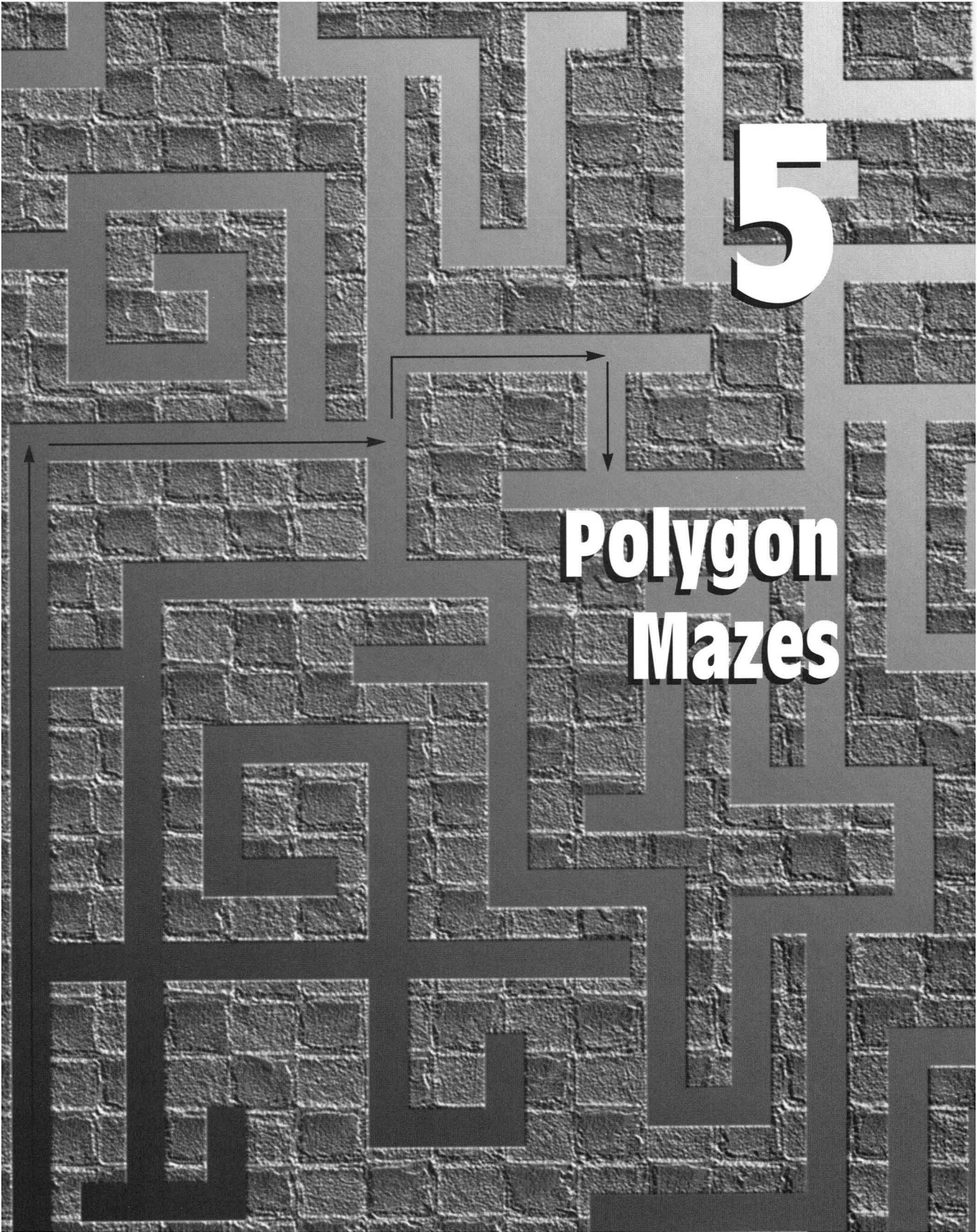
At this point, you may want to turn to expanded memory, or EMS, to loosen the crunch. We won't be discussing expanded memory in any detail in this book, but you can find information on it in a number of books on system-level PC programs. Basically, EMS provides your program with a standard method of accessing the memory in your machine beyond the 1 megabyte address barrier, via a buffer called a page frame located in high memory. You can call DOS to switch the contents of your computer's extended memory into this buffer 64 kilobytes at a time. (Older computers used special memory boards to provide expanded memory, but 386 and 486 computers — the machines that we're targeting with the programs in this book — use the extended memory beyond 1 megabyte to "emulate" these expanded memory boards, with the help of expanded memory managers such as EMM386, which comes with Windows and recent versions of DOS.) With the aid of expanded memory, you can greatly increase the number of bitmaps that your program is able to store.

There are alternatives to EMS, most notably XMS and the two protected mode standards, DPMI and VCPI. You can find information about these in the same sources that provide information about EMS. But XMS is a fairly slow way to move data to and from extended memory — essentially, it works like a fast RAM disk, copying the data back and forth a byte at a time — and the protected mode standards are not yet supported by the Borland C++ compiler. (At least, they weren't at the time this was written.) In the near future, one of the protected mode standards — probably VCPI, which is supported by Windows — will be the best way to access memory beyond the 640-kilobytes barrier. For the moment, though, EMS is probably your best alternative.

Flying Through the Maze

The final flaw in this maze engine — and the reason that we won't be expanding it any further in this book — is that, despite the vividly ray-traced bitmaps that we used in the graphics, the way in which the viewer must move around in the maze is inherently unrealistic. The viewer moves by jumps and spurts, going from the center of one maze square to the next without passing through the space in between. When rotating in place, the viewer can only take one of the four cardinal compass positions.

What we need to create an even more realistic maze engine is some method of incorporating flight simulator-style graphics techniques in our code, to create a maze animation with motion as realistic as that in a flight simulator. This is a big order, so big that we'll be devoting the rest of this book to finding a way to fulfill it. But fulfill it we will, as we move on into the world of polygon fill graphics, ray tracing, and the newest maze game programming technique of all, ray casting.


5
Polygon
Mazes

5

There are many different ways to render a three-dimensional, bitmapped world on the video display of a microcomputer. The simplest, if not the cheapest, is to hire an artist to paint three-dimensional bitmaps using a paint program such as Deluxe Paint or PC Paintbrush, then incorporate those bitmaps into your programs exactly as they were drawn. The results may well be stunning, if the artist is talented and experienced in the nuances of computer graphics. But your ability to use the bitmaps in a game program will be limited. You can only display the bitmapped image in the way that the artist drew it. You won't be able to turn it around and see what's on the other side.

The second easiest method, which we demonstrated in chapter 4, is to have an artist (or a rendering program, such as a ray tracer) create *parts* of an image, then slice and dice those pieces onto the screen so as to produce the illusion of a three-dimensional universe. Although the images produced by this method are usually somewhat less stunning than those an artist might create from whole cloth, they can be used in a wider variety of ways. With the slice and dice method, you actually *can* see what's on the other side of the image. But there are two drawbacks to this method. The first is that storing the bitmaps can require inordinate amounts of system memory, which in turn limits the variety of

bitmaps that you can incorporate into your game. The second is that the player's movements through the game world are constrained by the nature of the bitmaps. The player must move by quantum jumps from position to position, for the simple reason that there are no bitmaps showing what the world looks like from the intermediate positions.

In this chapter and the next, we'll look at a third method of generating 3D graphics in a game program. This method, which we'll refer to loosely as polygon graphics, has traditionally been the favored method for drawing landscapes in flight simulators, but until recently it has been little used in maze games, perhaps because polygon-based images are not especially realistic when seen from the viewpoint of a dungeon-crawling hero. Recent advances in CPU speed, however, have made it possible to combine polygon graphics with a technique called texture mapping to produce images that are far more realistic than those generated with earlier polygon-fill techniques.

We'll discuss the rudiments of polygon drawing on the IBM-PC and compatibles in this and the following chapter. Then we'll use polygon-based techniques to create a maze game engine that produces images almost as striking as those in chapter 4, while using up only a fraction as much memory. Finally, before we move on to more advanced techniques in chapter 8, we'll discuss how these techniques could be used to produce a fully three-dimensional dungeon, through which the player can move smoothly and realistically.

Filled and Unfilled Polygons

A polygon is a shape defined by three or more edges, like the triangle in Figure 5-1 and the square in Figure 5-2. There's no requirement that these edges be of equal length, as you can see by looking at the five-sided polygon in Figure 5-3. The point at which two edges come together is called a vertex. Because a polygon is always completely enclosed — that is, the edges form a continous border around the polygon, with the last edge linked to the first — the number of edges in a polygon will be the same as the number of vertices.

Drawing a polygon on the video display of a computer isn't particularly difficult. The only information we would need to store about the polygon or polygons to be drawn is the location on the computer display of the vertices. We could then link these vertices with lines by calling the line-drawing function from chapter 2. Listing 5-1 contains a short program that draws a four-sided polygon on the mode 13h display. The result of running the program is shown in Figure 5-4.

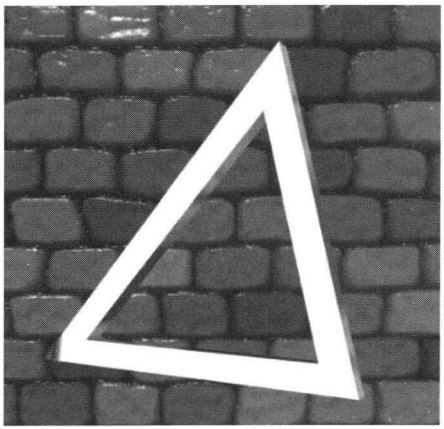


Figure 5-1 A triangle is an example of a simple polygon

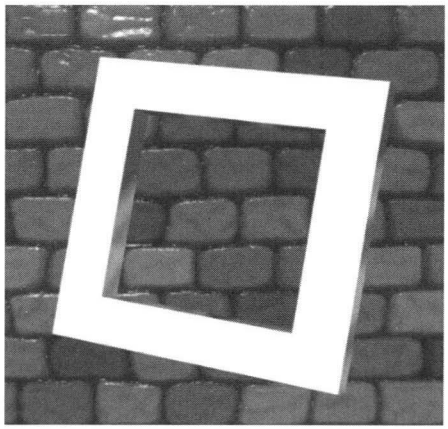


Figure 5-2 And so is a square

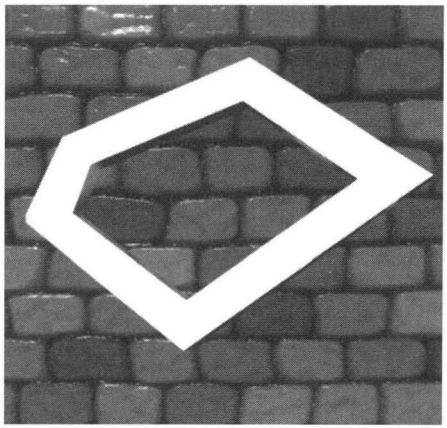
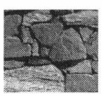


Figure 5-3 A polygon with five irregular sides



Listing 5-1 POLYGON.CPP

```
// POLYGON.CPP Version 1.0
// Draws an unfilled polygon on the mode 13h display
//
// Written by Christopher Lampton
//   for Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <conio.h>
```

continued on next page

continued from previous page

```
#include  <dos.h>
#include  "screen.h"
#include  "bresnham.h"

void main(int argc,char *argv[])
{
  // Create pointer to video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Put display in mode 13h:
  setmode(0x13);

  // Clear display:
  cls(screen);

  // Draw polygon:
  linedraw(17,24,107,68,15,screen);
  linedraw(107,68,190,79,15,screen);
  linedraw(190,79,150,180,15,screen);
  linedraw (150,180,17,24,15,screen);

  while (!kbhit());

  // Restore old video mode:
  setmode(oldmode);
}
```

The polygon drawn by the program in Listing 5-1 is an unfilled polygon — that is, it has a visible border but nothing inside that border. With only a few exceptions, the polygons used in games depicting three-dimensional

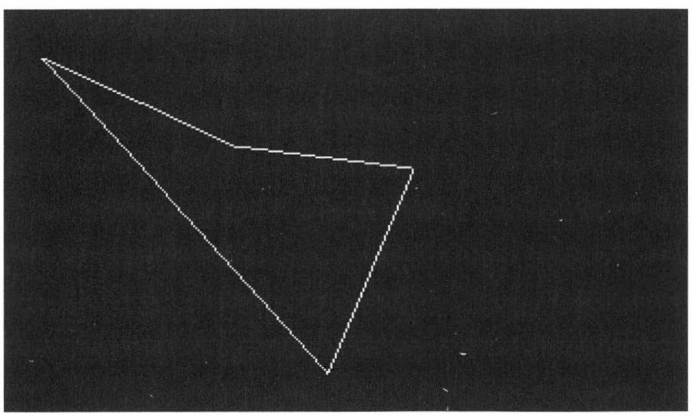


Figure 5-4 The screen output from the program POLYGON.CPP

environments are *filled* polygons. In a filled polygon, there is something inside the border. Just what that something is may vary from game to game.

In the simplest type of filled polygon graphics, the polygon is filled with pixels of a single color. This gives the polygon the look of solidity, as though it were a solid object rather than a collection of pixels on a video display. Several such polygons can be put together to form the three-dimensional image of an entire object, such as a building or an airplane. And, in fact, such solid-color filled polygons were until recently the basis of almost all the flight simulators on the market. This technique is so popular that images produced in this manner are usually referred to simply as polygon-fill graphics.

There are other ways to fill a polygon, which we'll talk about in a moment. For the moment, we'll take a closer look at polygon-fill graphics to see if they might be useful to us in creating a realistic maze game.

Filling a Polygon

Drawing a filled polygon on the video display of a microcomputer is a bit more difficult than drawing an unfilled polygon. In this chapter we'll simplify this task by restricting the shape of our polygons to those that will be useful in a maze game. What sort of shapes do we want to draw? Pretty much the same shapes as we were drawing in chapter 4: four-sided wall segments. This, in fact, is the only type of polygon that most maze game engines can draw. (The Ultima Underworld games appear to be an exception to this rule.) More complex objects are depicted with predrawn bitmaps, which are superimposed on top of these polygons. (We'll talk more about superimposing predrawn bitmaps in a later chapter.)

So all we really need to draw are four-sided polygons. However, this doesn't mean that we'll only be drawing rectangles. We'll also need to rotate these four-sided polygons, so that we can depict them as they would appear from various angles. This means we can't restrict ourselves to drawing only those polygons in which the edges meet at right angles. Which is too bad, since such polygons are quite easy to draw.

However, we can still simplify our task by restricting the rotations of these polygons to those rotations that take place around the polygon's local y axis. This means that the polygons we draw can rotate like tops (see Figure 5-5), but cannot roll head over heels (Figure 5-6), or spin like a propeller (Figure 5-7). This shouldn't noticeably inconvenience the player, since most movement in a maze game is one-dimensional. There's no reason, for instance, that the player will need to tilt his or her virtual head to one side, causing the polygons to appear to rotate on their z axes. Nor will the player need to do forward rolls down the maze corridor, causing the polygons to appear to rotate on their x axes. We do,

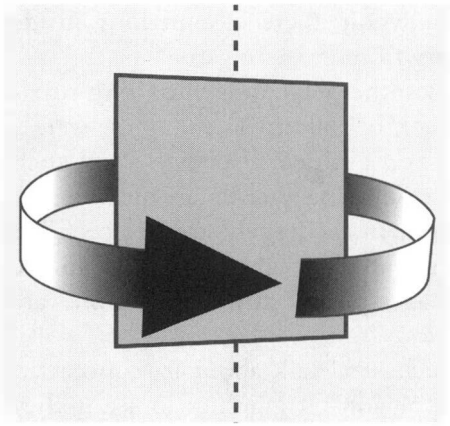


Figure 5-5 A polygon rotating like a top around its local y axis

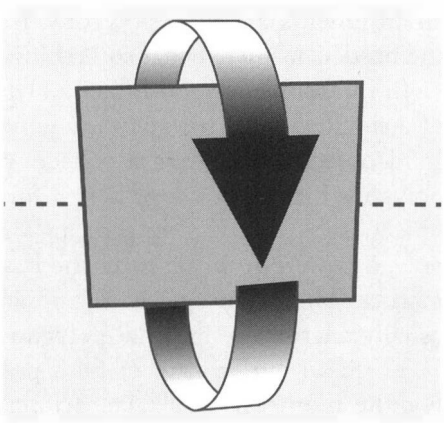


Figure 5-6 A polygon rolling head over heels around its local x axis

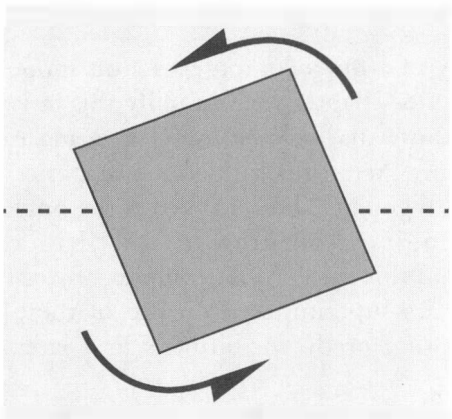


Figure 5-7 A polygon spinning like a propeller around its local z axis

however, want to give our players the ability to move up and down within the maze, simulating crouching or standing on top of raised platforms.

What we'll be drawing, then, are polygons representing wall segments that appear to rotate on their local y axes but that can also be depicted in such a way that the player seems to be looking at them from a low vantage point or a high vantage point. Thus we'll need a polygon-drawing function that will draw filled four-sided polygons in which the right and left edges are perfectly vertical but in which the other two edges can take any orientation whatsoever (except for perfectly vertical, since that would make them parallel to the other two edges). Figures 5-8a through d show examples of the sort of polygons that we'll be drawing.

The polytype Structure

Before we can write a function that draws a polygon, we need to define a variable structure that will hold a description of the polygon to be drawn. This structure can then be passed to the polygon-drawing function as a parameter. All that we'll need to place in this structure are the x,y coordinates at which the vertices of the polygons will be drawn on the video display. Since we're restricting our polygon drawing to four-sided polygons, this can be handled with a pair of four element arrays, one for the x coordinates and one for the y coordinates. Here's the definition for the *polytype* structure, which we'll place in the file POLYDRAW.H:

```
struct polytype {
     int x[4];
     int y[4];
};
```

Polygon Clipping

There is one other thing that we need to take into account before we start writing our polygon-drawing function. In a maze game, all of the action takes place in a rectangular window on the video display. This window is sometimes called the viewport. In drawing a three-dimensional image of the maze, we must be sure

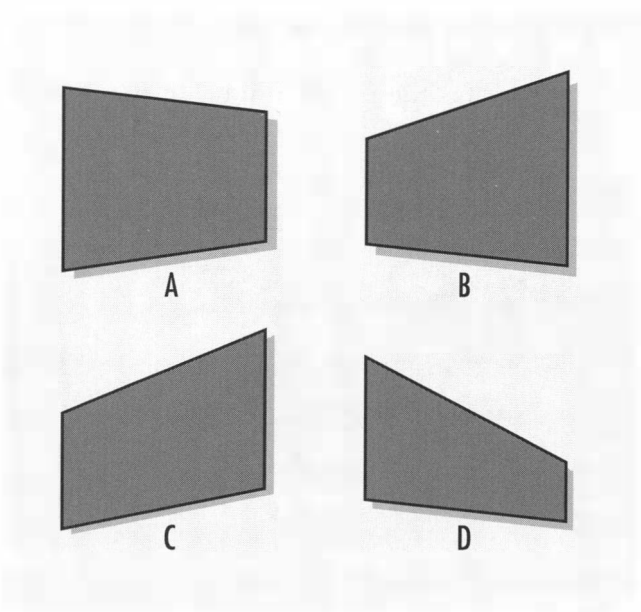


Figure 5-8a–d Four types of polygons that our polygon drawing function will be called on to draw

that we don't accidentally draw outside of the confines of this viewport. In earlier chapters, this has been a simple enough task. In chapter 4, for instance, we handled this by passing a pair of values to the maze-drawing function that defined the limits of the viewport, and we carefully kept all of our sliced and diced bitmaps inside those limits.

The task of keeping polygons inside the viewport is a bit more complex. In Figure 5-9a, for instance, a four-sided polygon needs to be clipped against the side of a viewport. Once clipped (as in Figure 5-9b), this four-sided polygon becomes a five-sided polygon. Thus, even though we have restricted the polygons that we'll be drawing to four-sided polygons, in practice these polygons might turn into five-sided or even six-sided polygons after being clipped at the edge of the viewport.

There are two general ways of handling polygon clipping. We can write a function that will accept a polygon definition and which returns to us a version of that polygon definition that has been clipped against the four edges of the viewport. There are standard algorithms for doing this. However, they are useless to us if we confine ourselves to drawing four-sided polygons, because clipping may add extra edges to the polygon. The *polytype* structure wouldn't even be capable of holding a description of a five-sided polygon.

We don't want to abandon the simplicity of the *polytype* structure, so we'll use the second method of clipping polygons. This method places the clipping in the polygon-drawing function itself. The function will watch for polygons that need to be clipped against the viewport and will do the clipping on the fly. Although this complicates the polygon-drawing function, it simplifies the rest of the

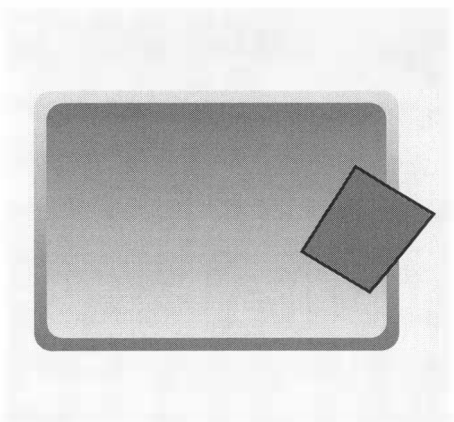


Figure 5-9a A four-sided polygon that needs to be clipped against the side of a viewport

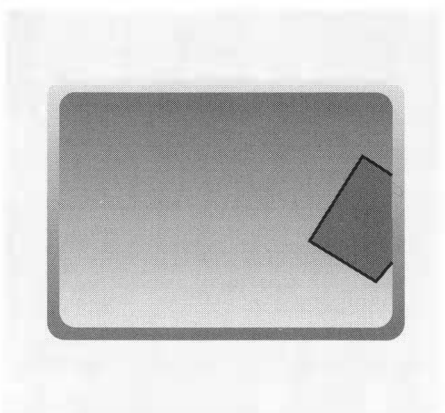


Figure 5-9b The four-sided polygon becomes a five-sided polygon after clipping

program. All we have to do when we draw a filled polygon is to call the polygon-drawing function, then pass it a description of the polygon in a *polytype* structure along with the dimensions and location of the viewport we wish the polygon to be clipped against.

A Polygon-Drawing Function

We'll call the polygon-drawing function *polydraw()*. Here's the prototype for the function:

```
void polydraw(polytype *poly,int leftx,int lefty,int rightx,
              int righty,int color,char *screen);
```

We'll place this prototype in the file POLYDRAW.H, along with the *polystruct* definition. We'll place the polydraw function itself in the file POLYDRAW.CPP.

A description of the polygon to be drawn is passed to *polydraw()* in the *polytype* structure *poly*. (Actually, a pointer to this structure is passed to *polydraw();* we don't want to waste the CPU's time by passing the entire structure.) To simplify our polygon-drawing algorithm, we will observe the convention that the coordinates of the upper-left corner of the polygon will be in *poly->x[0]* and *poly->y[0]*, with the remaining elements of the *x* and *y* arrays holding the coordinates of the remaining corners, moving clockwise around the polygon. Thus, *poly->x[1]* and *poly->y[1]* will contain the coordinates of the upper-right corner, *poly->x[2]* and *poly->y[2]* will contain the coordinates of the lower-right corner, and *poly->x[3]* and *poly->y[3]* will contain the coordinates of the lower-left corner. (See Figure 5-10.)

(Strictly speaking, we don't need to include the *poly->x[2]* and *poly->x[3]* fields in the polygon structure, since we're going to observe the convention that all polygons have perfectly vertical right and left edges and therefore the same x coordinates at the top and bottom vertices of that edge.)

The *leftx* and *lefty* parameters represent the x and y coordinates of the upper-left corner of the viewport that the polygon is to be clipped against. The *rightx* and *righty* parameters represent the coordinates of the lower-right corner of the viewport. The *color* parameter is the palette number of the color in which the polygon is to be drawn and the **screen* parameter is a pointer to the video display or to the buffer in which the drawing is to take place.

The Polygon-Drawing Algorithm

How will we go about drawing the polygon? The algorithm that we're going to use will break the polygon into a series of lines, each of which will be drawn on the video display with a *for()* loop. Figure 5-11 shows how this will work. Figure 5-11a shows an unfilled polygon drawn as a series of connected edges. Figure

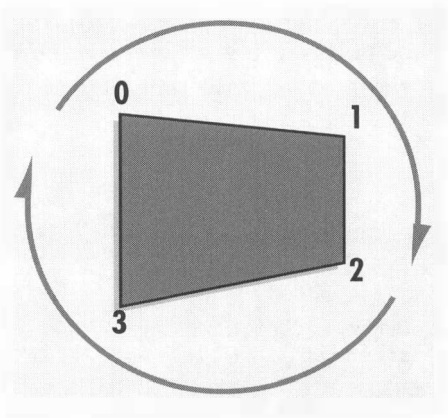


Figure 5-10 The coordinates of the vertices of the polygon will be specified in clockwise order starting at the upper-left corner

5-11b shows a filled version of the same polygon drawn as a series of vertical lines. What makes this drawing algorithm possible is the fact that we have limited ourselves to polygons with vertical left and right edges. The left and right edges will be treated as two of the lines in the polygon's fill.

Figuring out where to draw these two lines will be simplicity itself: The line for the left edge will be drawn vertically from coordinates *poly->x[0],poly->y[0]* to *poly->x[3],poly->y[3]* and the right edge will be drawn vertically from coordinates

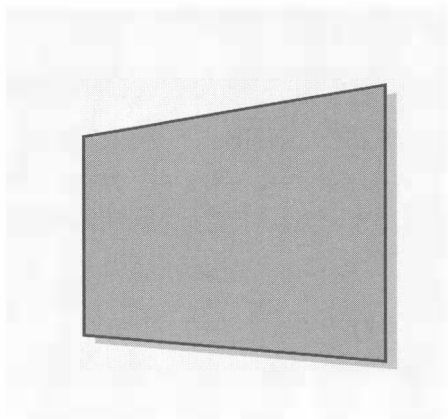


Figure 5-11a An unfilled polygon drawn as a series of connected edges

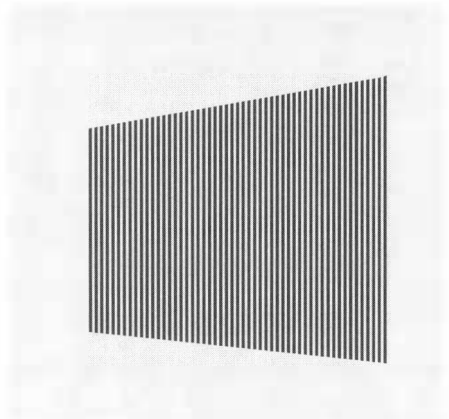


Figure 5-11b The same polygon drawn as a series of vertical lines

poly->x[1],poly->y[1] to *poly->x[2],poly->x[2]*. The real trick in drawing the polygon will be figuring out where the intervening lines will be drawn. It helps to remember that the top and bottom coordinates of these lines are all points on the top and bottom edges of the polygon. Thus, we can calculate these coordinates the same way we calculate the points on a line, using Bresenham's algorithm.

If you were paying attention back in chapter 2, you'll recall that Bresenham's algorithm uses a form of incremental division to calculate the changes in coordinates along a line. A value known as an error term is maintained and the difference between either the y or x coordinates of the endpoints (whichever was smaller) is added to this value and the value is compared with the length of the line on each iteration of the line-drawing loop. When the error term becomes greater than the length of the line in the other dimension, the pixel position is advanced in the direction represented by the error term. We can use that same trick to calculate the endpoints of the vertical lines making up the polygon, except that we'll need to maintain a *pair* of error terms — one for the top coordinates of the line and one for the bottom coordinates.

Clipping Against the Left Edge of the Viewport

Before we can perform any polygon drawing, we must perform some polygon clipping. However, in the name of efficiency, we are going to employ different methods for clipping against each edge. We could simplify our task greatly by having the polygon-drawing function step through all of the vertical lines that make up the polygon, checking before drawing to see if the line falls within the viewport. If the line is not in the viewport, it would simply not be drawn. Similarly, when drawing the individual pixels that make up the vertical line, the function could check to see if each pixel is outside the viewport and not draw it if it is. This method, however, would waste a lot of the CPU's valuable time, since it would require the function to step through a large number of lines and pixels that don't need to be drawn.

Fortunately, there's an alternate method that we can use to clip the polygon — or, rather, several alternate methods. During the initialization portion of the polygon-drawing function, we can check to see if any portion of the polygon extends beyond the left edge of the viewport. If it does, we can recalculate the coordinates of the upper-left and lower-left corners of the polygon to reflect the point at which they cross the edge. This method has a disadvantage too: It will require floating point math, which is notoriously slow. If the user has a math coprocessor in his or her computer, this method will still probably be faster than the first method; but if no math coprocessor is present, this method could well be slower. Since the majority of machines now have math coprocessors installed — the coprocessor comes standard with every 486DX chip, though not with the

486SX — we'll go with this second method. (In a later chapter, we'll discuss fixed point arithmetic, which can provide us with a high speed replacement for the slow floating point operations that we'll be using in this chapter. Readers interested in speeding up this program may want to rewrite it using fixed point math, which should run faster both on machines with coprocessors and without.)

We can check for a polygon that crosses the left edge of the video display by comparing the value of *poly->x[0],* the x coordinate of the upper-left corner of the polygon being pointed to by the parameter *poly,* with the value of *leftx,* the x coordinate of the upper-left corner (and therefore the common x coordinate of all points on the left edge) of the viewport, to see which is smaller:

```
if (poly->x[0]<leftx) {
```

If *poly->x[0]* is smaller than *leftx,* we know that the polygon extends past the left edge of the viewport. So we'll need to recalculate both *poly->x[0],poly->y[0]* and *poly->x[3],poly->y[3].* (Actually, we don't need to recalculate *poly->x[3],* since it should always be identical to *poly->x[0].*) Recalculating the x coordinates where the upper and lower edges of the polygon cross the left edge of the screen is simple enough. We can simply reset these coordinates to the x coordinate of the left edge, which is stored in *leftx.* Finding the y coordinates at which the edges cross the left edge is a bit tougher. We'll need to use the so-called point-slope equation to find these coordinates, which will in turn require some floating point (or fixed point) math.

Calculating the Slope

First we must calculate the slopes of the edges. You'll recall from chapter 2 that the slope of the line is the ratio of the change in y coordinates along the length of the line to the change in x coordinates. Thus, to obtain the slope, we must divide the change in the y coordinate by the change in the x coordinate. In fact, we'll need to do this twice, once for the upper edge of the polygon and once for the lower edge of the polygon, since each of these can have a different slope. We'll store the slope of the top edge in the variable *tslope* and the slope of the bottom edge in the variable *bslope,* like this:

```
tslope=(float)(poly->y[1]-poly->y[0])/(poly->x[1]-poly->x[0]);
bslope=(float)(poly->y[3]-poly->y[2])/(poly->x[0]-poly->x[1]);
```

As in chapter 2, we determine the change in x and y coordinates by subracting the initial x and y coordinates of the line from the ending x and y coordinates. You'll notice that we don't reference *poly->x[2]* and *poly->x[3]* when calculating the slope of the bottom edge, using instead the starting and ending points of the upper edge. In fact, we'll never reference the x coordinates of the lower edge

anywhere in the *polydraw()* function, using instead the identical x coordinates of the top edge. This allows us to safely skip some operations that reference the x coordinates of the bottom edge, making this function slightly more efficient. You'll also notice that we force the value of the division to floating point with the *(float)* cast. If we failed to do this, the result of the division would be an integer, even though *tslope* and *bslope* (declared earlier in the function) are floating point variables.

The Point-Slope Equation

Now that we have the slope, we can calculate the y coordinate of the edges using only the slope, the x coordinate of the edge to be clipped against, and the coordinates of one point on each line (hence the name "point-slope equation"). The general formula for this is:

```
new_y =  y + slope * (xmin - x)
```

where x and y are the coordinates of any point on the line, *xmin* is the x coordinate of the left edge, and *new_y* is the y coordinate of the point at which the line crosses the left edge. Using this formula, we can use *tslope* and *bslope* to find the new coordinates of the upper- and lower-left corners of the polygon:

```
     poly->y[0]=poly->y[0]+tslope*(leftx-poly->x[0]);
     poly->y[3]=poly->y[3]+bslope*(leftx-poly->x[0]);
     poly->x[0]=leftx;
}
```

You'll notice that this code ignores the x coordinate of the lower-left corner, using instead the identical x coordinate of the upper-left corner. And the new value of that coordinate isn't calculated here, since we won't be using it later. This ends the initial polygon clipping, though we'll be doing more in a moment. There's no need to check right away to see if the polygon extends past the other three edges of the viewport, since we'll be using different methods for clipping each of these. More about this in a moment.

The Drawing Begins

Since the x and y coordinates at which we are drawing will change as we traverse the polygon from left to right, we'll store the initial drawing coordinates in the integer variables *x* and *y*. Initially, these values will be equal to the coordinates of the upper-left corner:

```
int x=poly->x[0];
int y=poly->y[0];
```

Now we need to calculate the changes in the y coordinates along the top and bottom edges, for the variation on Bresenham's algorithm that we'll be using to draw those edges. We'll store the difference in y coordinates for the top line in the integer variable *topdiff*, and the difference in y coordinates for the bottom line in the integer variable *botdiff*:

```
int topdiff=poly->y[1]-poly->y[0];
int botdiff=poly->y[2]-poly->y[3];
```

We also need to calculate the height of the first vertical line to be drawn in the polygon. Since this line is at the far left edge of the polygon, its height will be the same as the height of the left edge:

```
int height=poly->y[3]-poly->y[0];
```

We also need to know the width of the polygon — that is, the number of vertical lines that we will be drawing. This value can be obtained by subtracting the x coordinate of the polygon's left edge from the x coordinate of the polygon's right edge and adding 1 to it:

```
int width=poly->x[1]-poly->x[0]+1;
```

Clipping Against the Right Edge

The value that we use here for the x coordinate of the left edge has already been clipped against the left edge of the viewport. However, the value that we are using for the x coordinate of the right edge has not yet been clipped. So we must check to see if the line needs clipping:

```
if (poly->x[1]>rightx)
```

If it does need clipping, we only need to revise the value of *width* to reflect the width of that portion of the polygon that will actually be drawn — that is, the portion inside the viewport. It's not necessary to recalculate the x and y coordinates at which the two edges cross the right edge of the viewport. Instead, we'll write the drawing code so that it stops drawing when it reaches that edge. Because we'll be using the variable *width* as part of the Bresenham code, we'll put the revised value in a new variable, which we'll call *vwidth*. This integer variable will be assigned a value equal to the distance between the right edge of the polygon and the right edge of the viewport if the two overlap. Otherwise, it will be set to the current value of *width*:

```
     vwidth=width-(poly->x[1]-rightx);
else vwidth=width;
```

The Error Terms

Bresenham's algorithm, as was noted in chapter 2, uses a value known as an error term to determine when it is time to shift the coordinates of the line in x or y dimension. We need to calculate the coordinates of the top and bottom edges, so we'll use a pair of integer variables to hold these terms. The one for the top edge will be called *toperror*, and the one for the bottom edge will be called *boterror:*

```
int toperror=0;
int boterror=0;
```

The Main Loop

Now we can begin drawing the vertical lines that will make up the polygon. How many vertical lines will we be drawing? One for every pixel in the visible width of the polygon, a value that we've already calculated and stored in the variable *vwidth*. We'll use a *for()* loop to step through the polygon's width:

```
for (int w=0; w<wwidth; w++) {
```

The next step is to draw a vertical line from the top edge of the polygon (the coordinates of which are stored in *x* and *y*) to the bottom edge (the distance to which is stored in the variable *height*). First, however, we must clip the vertical line against the top and bottom of the viewport. To clip the top of the line, we can compare the value of *y* with the y coordinate of the top edge of the viewport, which is stored in *lefty:*

```
if (y<lefty) {
```

If the line extends above the top edge, we'll simply start drawing at the top edge. Thus we'll assign the y coordinate of the top viewport edge to the integer variable *vy*, which will represent the actual point at which drawing should begin:

```
vy=lefty;
```

We'll also need to adjust the height of the line to clip out the portion that extends above the top edge of the viewport. We'll store the new height in the integer variable *vheight:*

```
vheight=height-(lefty-y);
}
```

If the line doesn't need to be clipped, we'll simply pass the current values of *y* and *height* to *vy* and *vheight*:

```
else {
     wy=y;
     wheight=height;
}
```

The bottom end of the vertical line also needs to be clipped against the bottom edge of the viewport. Should it need clipping, all we have to do is adjust the value of the *vheight* variable to reflect the clipped height of the line:

```
if ((wy+wheight)>righty)
     wheight=righty-wy;
```

Finally, before the line can be drawn, we need to calculate the video memory address corresponding to the screen coordinates *x,vy:*

```
unsigned int ptr=wy*320+x;
```

Drawing the Line

The actual drawing of the line can be handled by another *for()* loop:

```
for(int h=0; h<wheight; h++) {
```

The variable *ptr* has been set to the offset in video memory (or within an offscreen buffer) at which the next pixel is to be drawn. To draw the pixel, we need only store the value of the *color* parameter at that location:

```
screen[ptr]=color;
```

We must then advance *ptr* to the address of the next pixel to be drawn. Since we are drawing a perfectly vertical line, we need simply to add 320 — the width of the mode 13h display — to *ptr:*

```
ptr+=320;
}
```

And that's all there is to the loop that draws the vertical lines that make up the polygon. This innermost loop has deliberately been kept tight, because it's where most of the CPU cycles are going to go during the drawing of the polygon.

The Final Details

Several things must be done before we can move on to drawing the *next* vertical line, however. First, we must advance the variable *x,* which is used in the calculation of video memory offsets, to point to the x coordinate of the next vertical line:

```
x++;
```

Then we must decide whether to move the *y* offset of the top of the line up or down. As in our earlier Bresenham-based line-drawing function, this is done with the aid of an error term:

```
toperror+=abs(topdiff);
```

You'll recall that the variable *topdiff* represents the amount of change in the *y* dimension from the first end of the top edge to the second. This value can be positive or negative, depending on whether the line slopes up or down from left to right. When adding this value to the error term, though, we ignore the sign and add the absolute value of *topdiff* to *toperror*. If the resulting sum is greater than the width of the polygon, we need to adjust the y coordinate of the edge:

```
while (toperror>=width) {
```

You'll notice that we use a *while()* statement rather than an *if()* statement here. This covers the possibility that the change in y coordinates along the top edge of the line may be larger than the change in x coordinates, in which case we'll need to advance the y coordinate by several pixels rather than just one. First, though, we subtract the value of *width* to reset the error term:

```
toperror-=width;
```

Then we advance the y coordinate of the top edge of the polygon. Which direction do we advance it in? That depends on whether *topdiff* is negative or positive. If positive, the line must slant downward, so we must increase the y coordinate by 1:

```
if (topdiff>0) {
     y++;
     Ñheight;
}
```

We also subtract 1 from the height of the line so that this instruction doesn't move the y coordinate of the bottom edge down as well.

If *topdiff* is negative, we do the opposite:

```
     else {
          --y;
          height++;
     }
}
```

If *topdiff* is equal to 0, then the top edge is perfectly horizontal and we do nothing at all, since the value of the y coordinate never changes along a horizontal line. Notice that if *topdiff* is larger than *width* — that is, if the change in y is greater than the change in x — this *while()* loop will repeat more than once, advancing the y coordinate by more than one position.

We must repeat this sequence for the bottom edge:

```
boterror+=abs(botdiff);
while (boterror>=width) {
     boterror-=width;
     if (botdiff>0) height++;
```

```
                else --height;
            }
        }
}
```

When dealing with the bottom edge, we don't need to advance the value of *y*, but we do need to change the height of the line, either upward or downward as appropriate. Now we are ready to draw the next line, which completes the polygon-drawing loop. The full text of the *polydraw* function is in Listing 5-2.

The polydraw() Function

Listing 5-2 The polydraw() function

```
#include <stdio.h>
#include <math.h>
#include "polydraw.h"

// Function to draw a polygon defined by POLYTYPE parameter
// POLY in color COLOR clipped against a viewport with upper
// lefthand coordinates of LEFTX,LEFTY and lower righthand
// coordinates of RIGHTX,RIGHTY.

void polydraw(polytype *poly,int leftx,int lefty,int rightx,
                int righty,int color,char *screen)
{
  float tslope,bslope;
  int wwidth,wy,wheight;

  // If polygon crosses left side of viewport, clip it:
  if (poly->x[0]<leftx) {

  // Calculate slopes for top and bottom edges:
    tslope=(float)(poly->y[1]-poly->y[0])/(poly->x[1]-poly->x[0]);
    bslope=(float)(poly->y[3]-poly->y[2])/(poly->x[0]-poly->x[1]);

    // Find new endpoints for clipped lines:
    poly->y[0]=poly->y[0]+tslope*(leftx-poly->x[0]);
    poly->y[3]=poly->y[3]+bslope*(leftx-poly->x[0]);
    poly->x[0]=leftx;
  }

    // Initialize x,y coordinates for polygon drawing:
    int x=poly->x[0];
    int y=poly->y[0];
```

```
// Calculate the change in y coordinates for top
// and bottom edges:
  int topdiff=poly->y[1]-poly->y[0];
  int botdiff=poly->y[2]-poly->y[3];

// Initialize height and width of clipped polygon:
  int height=poly->y[3]-poly->y[0];
  int width=poly->x[1]-poly->x[0]+1;

// Clip polygon width against right side of viewport:
  if (poly->x[1]>rightx)
    wwidth=width-(poly->x[1]-rightx);
    else wwidth=width;

// Initialize top and bottom error terms:
  int toperror=0;
  int boterror=0;

// Loop across width of polygon:
  for (int w=0; w<wwidth; w++) {

  // If top of current vertical line is outside of
  // viewport, clip it:
    if (y<lefty) {
       wy=lefty;
       wheight=height-(lefty-y);
     }
    else {
      wy=y;
      wheight=height;
  }
  if ((wy+wheight)>righty)
    wheight=righty-wy;

  // Point to video memory offset for top of line:
  unsigned int ptr=wy*320+x;

  // Loop through the pixels in the current vertical
  // line, advancing PTR to the next row of pixels after
  // each pixel is drawn.
  for(int h=0; h<wheight; h++) {
     screen[ptr]=color;
     ptr+=320;
  }

  // Advance x coordinate to next vertical line:
  x++;

  // Is it time to move the top edge up or down?
  toperror+=abs(topdiff);
  while (toperror>=width) {
```

continued on next page

continued from previous page

```
       // If so move it up...
        toperror-=width;
        if (topdiff>0) {
          y++;
         --height;
        }

       // or down.
        else {
          --y;
           height++;
        }
    }

    // Is it time to move the bottom edge up or down?
      boterror+=abs(botdiff);
      while (boterror>=width) {

       // If so, move it:
     boterror-=width;
     if (botdiff>0) height++;
     else --height;
      }
   }
}
```

The POLYDEMO Program

For a look at the *polydraw()* function in action, we'll write a short demonstration program that will draw a polygon on the display. We'll use the same setup that we've used in earlier chapters for turning on mode 13h, saving and restoring the old video mode, establishing an offscreen video buffer, and so forth. Once the initialization is complete, we'll plug the appropriate values into a variable of type *polytype* to describe a polygon with an upper edge extending from coordinates 50,90 to coordinates 250,10 and a lower edge extending from coordinates 50,100 to coordinates 250,120:

```
poly.x[0]=50;
poly.y[0]=90;
poly.x[1]=250;
poly.y[1]=10;
poly.x[2]=250;
poly.y[2]=120;
poly.x[3]=50;
poly.y[3]=110;
```

Then we'll pass this structure to the polygon-drawing function:

```
polydraw(&poly,0,0,319,199,1,screen);
```

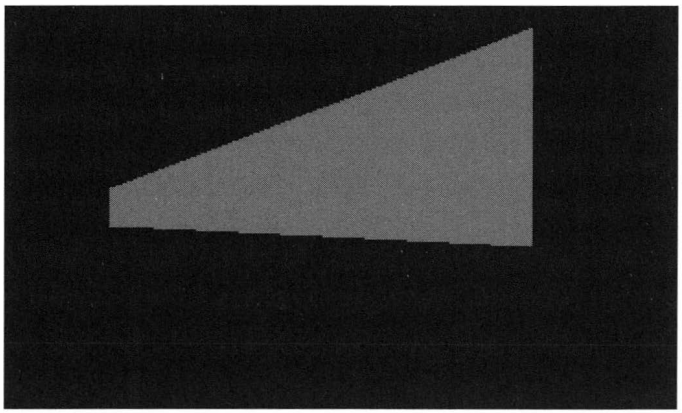
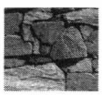


Figure 5-12 The video output from the program POLYDEMO.CPP

And that's all there is to drawing a filled polygon. The full text of the program POLYDEMO.CPP is shown in Listing 5-3 and the video output is in Figure 5-12.



Listing 5-3 The POLYDEMO.CPP program

```
#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  "screen.h"
#include  "polydraw.h"
#include  "bresnham.h"

void main()
{
  // Declare polygon structure:
  polytype poly;

  // Create pointer to video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Put display in mode 13h:
  setmode(0x13);

  // Clear display:
  cls(screen);
```

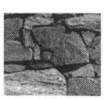
continued on next page

continued from previous page

```
  // Put polygon data in POLYTYPE structure:
  poly.x[0]=50;
  poly.y[0]=90;
  poly.x[1]=250;
  poly.y[1]=10;
  poly.x[2]=250;
  poly.y[2]=120;
  poly.x[3]=50;
  poly.y[3]=110;

  // Draw the polygon:
  polydraw(&poly,0,0,319,199,1,screen);

  // Hold polygon on screen until keystroke
  while (!kbhit());

  // Restore old video mode:
  setmode(oldmode);

}
```

The POLYCLIP Program

Ah, but what about the code we placed in the function to clip the polygon against a window? How do we know that it works correctly? To find out, we'll create a slight variation on POLYDEMO.CPP that will draw a box on the screen representing a viewport, then clip the polygon to that viewport. To draw the box, we'll resurrect the *drawbox()* function that we used in chapters 2 and 3. The complete listing of POLYCLIP.CPP appears in Listing 5-4 and a picture of its video output in Figure 5-13.



Listing 5-4 The POLYCLIP.CPP program

```
#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  "screen.h"
#include  "polydraw.h"
#include  "bresnham.h"

void drawbox(int leftx,int lefty,int rightx,int righty,
               char *screen);

void main()
{
  // Declare polygon structure:
```

```
  polytype poly;

  // Create pointer to video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Put display in mode 13h:
  setmode(0x13);

  // Clear display:
  cls(screen);

  // Draw box on display:
  drawbox(70,50,200,130,screen);

  poly.x[0]=50;
  poly.y[0]=90;
  poly.x[1]=250;
  poly.y[1]=10;
  poly.x[2]=250;
  poly.y[2]=120;
  poly.x[3]=50;
  poly.y[3]=110;
  polydraw(&poly,71,51,199,129,1,screen);

  // Hold polygon on screen until keystroke
  while (!kbhit());

  // Restore old video mode:
  setmode(oldmode);

}

void drawbox(int leftx,int lefty,int rightx,int righty,
       char *screen)
{
  linedraw(leftx,lefty,rightx,lefty,15,screen);
  linedraw(rightx,lefty,rightx,righty,15,screen);
  linedraw(rightx,righty,leftx,righty,15,screen);
  linedraw(leftx,righty,leftx,lefty,15,screen);
}
```

A Polygon Maze

It's not hard to plug this polygon-drawing function into the maze-drawing program that we created in chapter 4. Most of the necessary changes will be in the *drawmaze()* function. We'll also need to recast the array *matrix[]*, which

represents the screen y coordinates at which the horizontal and vertical walls of the maze cross one another on the display to take into account points that are just off the visible edge of the viewport:

```
int matrix[VIEW_LIMIT+2][VIEW_LIMIT*2+2]={
     {187,187,187,187,187,  0,  0,  0,  0,  0},
     {187,187,187,238,143, 47,-49,  0,  0,  0},
     {187,187,187,179,123, 66, 10,  0,  0,  0},
     {187,187,187,158,117, 73, 31,  0,  0,  0},
     {187,187,177,144,111, 78, 45, 12,  0,  0},
     {187,185,159,133,107, 82, 56, 30,  4,  0}
};
```

We'll also need to create two new arrays, which represent the x coordinates of the tops and bottoms of the polygons viewed at various distances:

```
int top[VIEW_LIMIT+2]={-11,20,36,42,46,49};
int bottom[VIEW_LIMIT+2]={131,99,83,77,73,70};
```

Because some of the polygon edges will actually extend beyond the limits of the viewport, we have included values in all of these arrays that exceed the limits of our viewport. All coordinates are relative to the upper-left corner of the viewport, which is 188 pixels wide, so all negative numbers and numbers larger than 187 fall outside of the viewport.

A New Maze-Drawing Function

The new maze-drawing function will be much like the old one. We'll use the same recursive algorithm for determining what walls need to be drawn as we used in chapter 4. (You might want to reread that chapter if you're having trouble remembering how the algorithm works.) Almost all of the changes concern the function calls that perform the actual drawing. In the bitmapped version that we developed in chapter 4, the *drawslice()* function was called to copy a slice from the appropriate bitmap onto the display. These will be replaced by calls to the *polydraw()* function.

Before we can call *polydraw(),* we must put a description of the polygon to be drawn in a variable of type *polytype.* The necessary values can be taken from the *matrix, top,* and *bottom* arrays. Here is how we draw the view through the lefthand portion of the current maze square, as required by the recursive algorithm that we're using:

```
poly.x[0]=matrix[yshift][xshift+VIEW_LIMIT+1]+XWINDOW;
poly.y[0]=top[yshift]+YWINDOW;
poly.x[1]=matrix[yshift+1][xshift+VIEW_LIMIT+1]+XWINDOW;
poly.y[1]=top[yshift+1]+YWINDOW;
```

```
poly.y[2]=bottom[yshift+1]+YWINDOW;
poly.y[3]=bottom[yshift]+YWINDOW;
```

Then we call *polydraw()* to draw the polygon:

```
polydraw(&poly,vleft+XWINDOW,YWINDOW,vright+XWINDOW,
            YWINDOW+WHEIGHT,2,screen);
```

As in the previous version, we pass the x coordinates of the right and left edges of the current viewing window within the viewport, contained in the *vleft* and *vright* coordinates. (See chapter 4 for a more detailed explanation of what these values represent.) Now, however, we must also pass the y coordinates of the top and bottom of the viewport. To all of these values, we must add the offsets of the view window (which are contained in the XWINDOW and YWINDOW constants) to translate our view window relative coordinates into absolute coordinates on the video display.

We repeat this process to draw the view over the middle portion of the current square:

```
poly.x[0]=matrix[yshift+1][xshift+VIEW_LIMIT+1]+XWINDOW;
poly.y[0]=top[yshift+1]+YWINDOW;
poly.x[1]=matrix[yshift+1][xshift+VIEW_LIMIT]+XWINDOW;
poly.y[1]=top[yshift+1]+YWINDOW;
poly.y[2]=bottom[yshift+1]+YWINDOW;
poly.y[3]=bottom[yshift+1]+YWINDOW;
polydraw(&poly,vleft+XWINDOW,YWINDOW,vright+XWINDOW,
            YWINDOW+WHEIGHT,2,screen);
```

and the view over the righthand portion of the current square:

```
poly.x[0]=matrix[yshift+1][xshift+VIEW_LIMIT]+XWINDOW;
poly.y[0]=top[yshift+1]+YWINDOW;
poly.x[1]=matrix[yshift][xshift+VIEW_LIMIT]+XWINDOW;
```

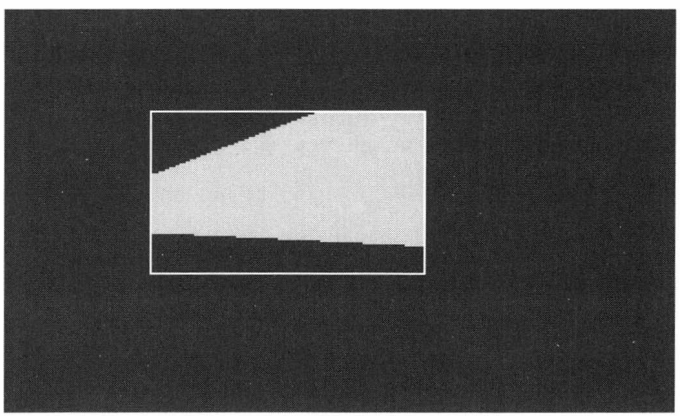


Figure 5-13 The video output from the program POLYCLIP.CPP

```
poly.y[1]=top[yshift]+YWINDOW;
poly.y[2]=bottom[yshift]+YWINDOW;
poly.y[3]=bottom[yshift+1]+YWINDOW;
polydraw(&poly,vleft+XWINDOW,YWINDOW,vright+XWINDOW,
           YWINDOW+WHEIGHT,2,screen);
```

The New drawmaze() Function

The text of the polygon-based version of the *drawmaze()* function appears in Listing 5-5:

Listing 5-5 The polygon-based drawmaze() function

```
void drawmaze(int xshift,int yshift,int lview,int rview,char *screen)

// Recursive function to draw left, right and center walls for the
// maze square at POS.X+XSHIFT, POS.Y+YSHIFT, as viewed from POS.X,
// POS.Y, within the screen window bordered by horizontal coordinates
// LVIEW and RVIEW on the left and right respectively. When called
// with XSHIFT and YSHIFT set to 0, draws entire maze as viewed
// from POS.X,POS.Y out to VIEW_LIMIT squares.

{
  int vleft,vright,vwidth;
  polytype poly;

  // Return if we've reached the recursive limit:
  if ((abs(xshift) > VIEW_LIMIT) || (abs(yshift) > VIEW_LIMIT))
     return;

  // Calculate coordinates of square to left:
  int sx = pos.x + (xshift+1) * left[dir].x + yshift * forward[dir].x;
  int sy = pos.y + (xshift+1) * left[dir].y + yshift * forward[dir].y;

  // Get left-right extent of left part of view:
  vleft=lview;
  vright=matrix[yshift+1][xshift+VIEW_LIMIT+1];
  if (vright>rview) vright=rview;
  vwidth=vright-vleft;

  if (vwidth>0) {

    // If the square to the left is occupied,
    // draw right side of cube:
    if (maze[sx][sy]) {
      poly.x[0]=matrix[yshift][xshift+VIEW_LIMIT+1]+XWINDOW;
      poly.y[0]=top[yshift]+YWINDOW;
      poly.x[1]=matrix[yshift+1][xshift+VIEW_LIMIT+1]+XWINDOW;
      poly.y[1]=top[yshift+1]+YWINDOW;
```

```
    poly.y[2]=bottom[yshift+1]+YWINDOW;
    poly.y[3]=bottom[yshift]+YWINDOW;
    polydraw(&poly,vleft+XWINDOW,YWINDOW,vright+XWINDOW,
              YWINDOW+WHEIGHT,2,screen);
  }

  // Else call function recursively to draw the view from
  // next square to the left:
  else drawmaze(xshift+1,yshift,vleft,vright,screen);

}

// Calculate coordinates of square directly ahead:
sx = pos.x + xshift * left[dir].x + (yshift+1) * forward[dir].x;
sy = pos.y + xshift * left[dir].y + (yshift+1) * forward[dir].y;


// Get left-right extent of center part of view:
vleft=matrix[yshift+1][xshift+VIEW_LIMIT+1];
vright=matrix[yshift+1][xshift+VIEW_LIMIT];
if (vleft<lview) vleft=lview;
if (vright>rview) vright=rview;
vwidth=vright-vleft;

if (vwidth>0) {

  // If the square directly ahead is occupied,
  // draw front of cube:
  if (maze[sx][sy]) {
    poly.x[0]=matrix[yshift+1][xshift+VIEW_LIMIT+1]+XWINDOW;
    poly.y[0]=top[yshift+1]+YWINDOW;
    poly.x[1]=matrix[yshift+1][xshift+VIEW_LIMIT]+XWINDOW;
    poly.y[1]=top[yshift+1]+YWINDOW;
    poly.y[2]=bottom[yshift+1]+YWINDOW;
    poly.y[3]=bottom[yshift+1]+YWINDOW;
    polydraw(&poly,vleft+XWINDOW,YWINDOW,vright+XWINDOW,
      YWINDOW+WHEIGHT,2,screen);
  }

  // Else call function recursively to draw the view from
  // next square forward:
  else drawmaze(xshift,yshift+1,vleft,vright,screen);
}

// Calculate coordinates of square to right:
sx = pos.x + (xshift-1) * left[dir].x + yshift * forward[dir].x;
sy = pos.y + (xshift-1) * left[dir].y + yshift * forward[dir].y;

// Get left-right extent of right part of view:
vleft=matrix[yshift+1][xshift+VIEW_LIMIT];
if (vleft<lview) vleft=lview;
vright=rview;
```

continued on next page

continued from previous page

```
  vwidth=vright-vleft;

  if (vwidth>0) {

    // If the square to the right is occupied,
    // draw left side of cube:
    if (maze[sx][sy]) {
      poly.x[0]=matrix[yshift+1][xshift+VIEW_LIMIT]+XWINDOW;
      poly.y[0]=top[yshift+1]+YWINDOW;
      poly.x[1]=matrix[yshift][xshift+VIEW_LIMIT]+XWINDOW;
      poly.y[1]=top[yshift]+YWINDOW;
      poly.y[2]=bottom[yshift]+YWINDOW;
      poly.y[3]=bottom[yshift+1]+YWINDOW;
      polydraw(&poly,vleft+XWINDOW,YWINDOW,vright+XWINDOW,
        YWINDOW+WHEIGHT,2,screen);
    }

    // Else call function recursively to draw the view from
    // next square to the right:
    else drawmaze(xshift-1,yshift,vleft,vright,screen);
  }
}
```

Touring the Polygon Maze

You can try out the complete POLYMAZE.CPP program by CDing to the POLYGONS directory from the distribution disk that came with this book and typing POLYMAZE. Once the program is running, you can walk through the polygon-based maze the same way that you moved through the bitmapped maze in chapter 4, by using the arrow keys on the keyboard or moving the mouse. However, you'll notice that this maze is nowhere near as visually interesting as the bitmapped maze in chapter 4 was. The walls are a uniform green color and it's difficult to determine where one corridor ends and a side corridor begins. The output from this program is shown in Figure 5-14.

Must we sacrifice realism in order to build a polygon-based maze? Not necessarily. There are several ways in which this maze could be made more interesting, though we don't have space in this book to try them all. One would be to assign different colors to different wall segments. The palette numbers for these colors could be placed in the *maze[]* array instead of the 0s and 1s that we've been using since chapter 2. Or we could change the intensity of the wall colors depending on how much light is falling on them. If we assume that all light comes from a torch or lantern being held by the player, walls at a greater distance from the player would be more dimly illuminated than walls that are closer, and walls turned at an angle to the player would be more dimly

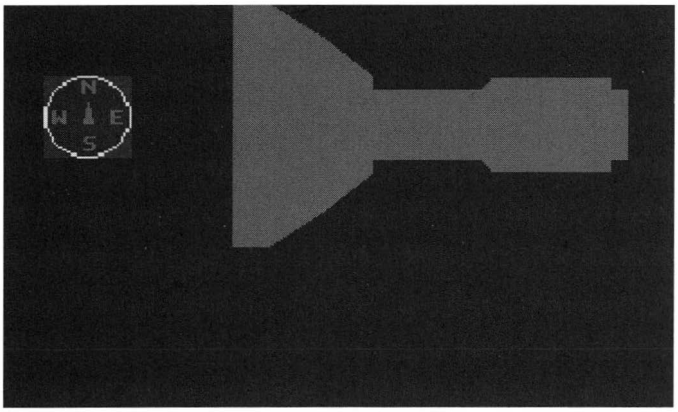
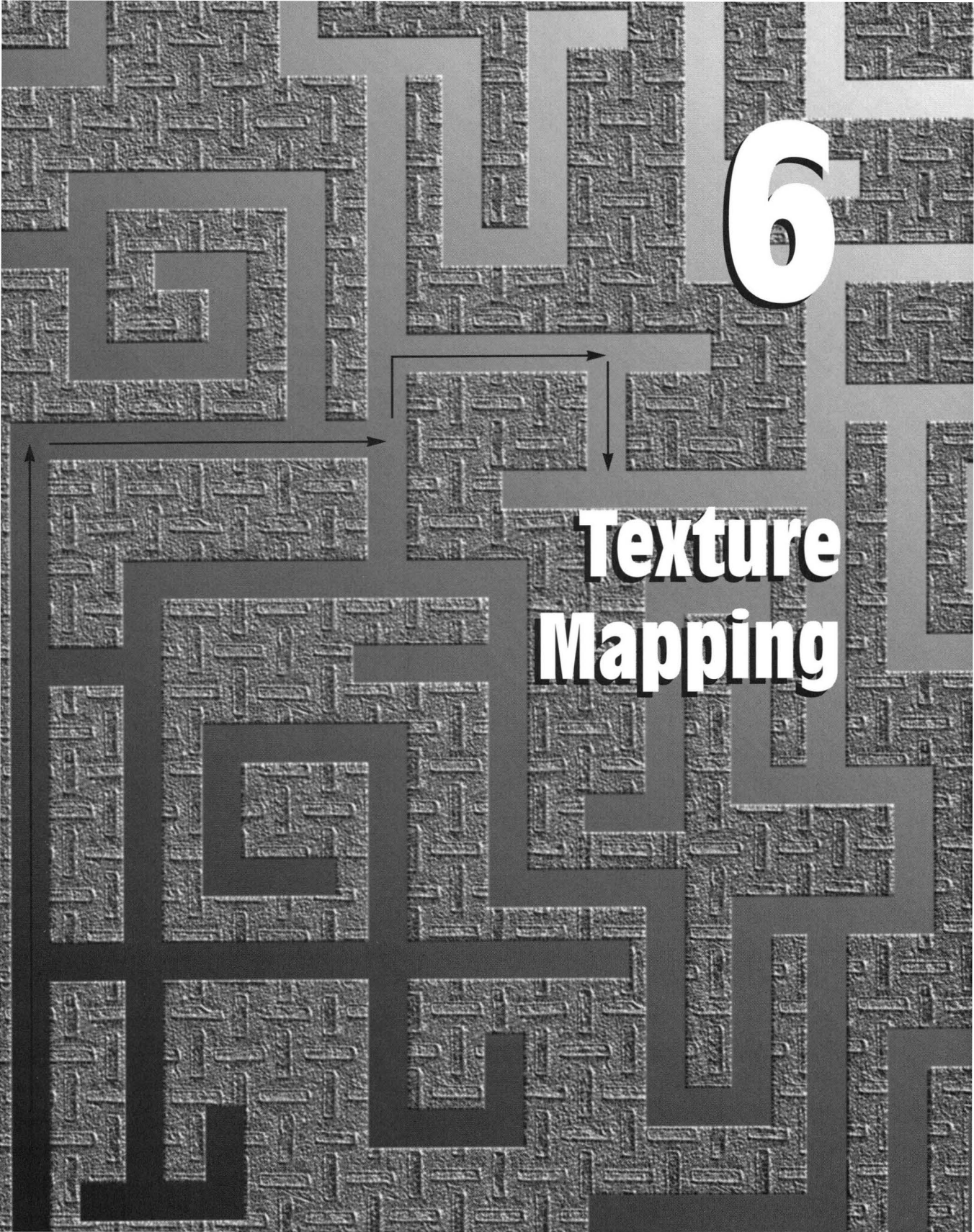
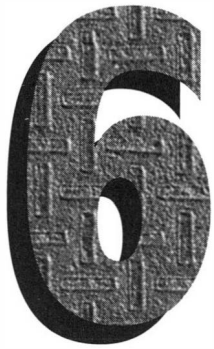


Figure 5-14 The video output from the program POLYMAZE.CPP

illuminated than walls that face the player head on. This technique is called *lightsourcing*, and we'll be studying it in more detail in a later chapter.

For now, we'll concentrate on yet another method of making the walls appear more interesting and realistic. This technique will allow us to place actual bitmaps on the walls and rotate those bitmaps along with the walls. In the next chapter, we'll take a closer look at *texture mapping*.


6
Texture
Mapping



Solid-colored polygons are boring. While they may be adequate for producing the distant scenery visible through the window of a simulated airplane, filled polygons just don't cut it as a method of drawing dungeon walls.

Fortunately, solid colors aren't the only thing that we can fill polygons with. We can also fill polygons with bitmaps. If we want to create the illusion of an ivy-covered brick wall, we can fill our polygon-based dungeon walls with bitmaps depicting bricks and ivy. If we want to create a window that looks out of the dungeon onto the rolling countryside, then we can fill a polygon with a bitmap of a window. And so on.

So how do we go about filling a polygon with a bitmap? Do we simply copy the bitmap byte by byte into the polygon's interior?

Not quite. In order to make the illusion complete, the bitmapped images must appear to grow smaller as the polygons recede into the distance and larger as we grow closer to them. And they must appear to rotate when the polygons rotate. Pulling off such an effect requires a bit more programming trickery than simply filling a polygon with a solid color. To graphics programmers, this trickery is known as *texture mapping*.

Two-and-a-Half-Dimensional Texture Mapping

Mapping a bitmap onto a three-dimensional polygon is, in a sense, a two-step process. The bitmap must be scaled for distance, as the polygon grows larger and smaller, and the bitmap must be rotated when the polygon rotates. Scaling is the easier of the two steps. To scale a bitmap, all you need to do is selectively add and remove pixels as you copy the bitmap to the video buffer, thus making the image appear to grow larger and smaller. Rotating the bitmap, on the other hand, can be a great deal more difficult, especially if the polygons on which you are mapping the bitmap can rotate on all three axes.

Fortunately, as we noted in the last chapter, the polygons in a maze game don't really need to rotate on all three axes. They only need to rotate on their y axes. This greatly simplifies the task of mapping bitmaps onto the polygons. In a sense, what we will be doing in this chapter isn't three-dimensional texture mapping. It is two-and-a-half-dimensional texture mapping.

The Two-Step Program

Because it is the simpler of the two steps, we'll start by discussing the scaling of bitmaps. As noted above, this is a relatively simple matter of adding and subtracting pixels from the bitmap, to make it appear to grow larger and smaller. The most difficult part of this process is deciding when to add pixels and when to subtract them.

In a few specific instances, this decision is quite easy to make. If we wanted to double the size of a bitmap, for instance, we would simply repeat every pixel (and every line of pixels) twice. And to halve the size of a bitmap, we would remove every other pixel (and every other line of pixels). But what do we do if we want to increase the size of a bitmap to, say, 137 percent of its original size?

The solution is to turn to our old friend incremental division. So far, we've used incremental division to draw lines and to find the upper and lower edges of a filled polygon. Now we're going to use it to scale bitmaps.

To demonstrate, we'll create a simple program that will scale the familiar bitmap depicted in Figure 6-1. You'll find this image in the texture directory as MONA.TGA. The background over which it will be displayed is in MONABG.TGA.

Bitmap Scaling

The bitmap-scaling demo will begin with a few constant definitions:

```
const FRAMES_PER_SECOND=2;  // Animation speed
const IMAGE_X=25;           // Location of bitmap in TGA FILE
const IMAGE_Y=21;
const IMAGE_WIDTH=76;       // Width of bitmap before scaling
const IMAGE_HEIGHT=96;      // Height of bitmap before scaling
const SCREENX=20;           // Screen location for scaled bitmap
```

We'll be creating a brief animation of sorts, so we first set the frames-per-second rate. The IMAGE_X and IMAGE_Y constants represent the x,y coordinates of the upper-left corner of the Mona image in the TGA file and the IMAGE_WIDTH and IMAGE_HEIGHT constants represent the width and height of that image. We'll pass this information to the *grab()* function (which we developed back in chapter 4) so that it can copy the bitmap into an array. Finally, we set a constant equal to the x coordinate at which we will be displaying the Mona bitmap on the screen. (We don't set a constant for the y coordinate because that coordinate will change as the bitmap is scaled.)

We'll also need a couple of TGA structures, one to hold the image to be scaled and one to hold the background image upon which the scaling will take place. We'll call these structures *mona* and *bg*:

```
tga_struct mona,bg
```

We also need an array in which to store the portion of the bitmap that we'll be "grabbing" from the *mona* structure:

```
char bitmap[IMAGE_WIDTH*IMAGE_HEIGHT];
```

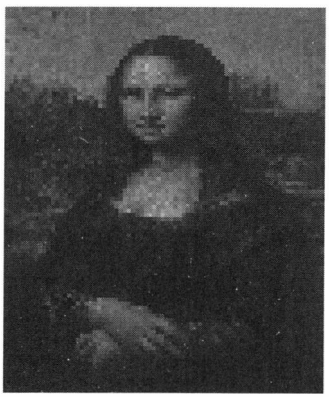


Figure 6-1 A familiar bitmapped image

The *main()* function will begin with the usual folderol: setting up a pointer to video memory and to an offscreen buffer, saving the old video mode, initializing the timer, and setting the video mode 13h. Then we'll load the two TGA images into their respective structures:

```
if (loadTGA("mona.tga",&tga)) {
   setmode(oldmode);
   exit(1);
}

if (loadTGA("monabg.tga",&bg)) {
   setmode(oldmode);
   exit(1);
}
```

We also need to set the palette, which raises an interesting question: Do we use the palette from MONA.TGA or the palette from MONABG.TGA? As it happens, either palette will do. Because these images are designed to be displayed together on the screen, it was necessary to use the same palette on both. For some tips on how to match palettes between two or more images using only the public domain PicLab program, see Appendix A. To set the palette in our program, we'll arbitrarily use the palette from MONA.TGA:

```
setpalette(mona.color_map);
```

We'll continue scaling the image until somebody presses a key:

```
while (!kbhit()) {
```

But before we begin, we'll want to copy the background image (which is contained in the *image* field of the *bg* structure) into the screen buffer:

```
for (unsigned int i=0; i<64000; i++)
  screen_buffer[i]=bg.image[i];
```

When we scale bitmaps, we'll need both to shrink the bitmaps and to enlarge them. To demonstrate how both of these operations are performed, we'll first reduce Mona to 10 percent of her original size, then gradually expand her to twice her normal dimensions. We'll use a *for()* loop to step the percentage of reduction from 10 percent to 200 percent, at 5 percent intervals:

```
for (int percent=10; percent<=200; percent+=5) {
```

This is a good spot to insert a frame delay, so that the scaling doesn't take place *too* quickly. We'll also reset the *lastframe* variable (which we've used in earlier programs) to indicate the time at which the next frame begins:

```
  while ((clock()-lastframe)<ticks_per_frame) ;
lastframe=clock();
```

The Vertical Error Term

We'll draw the image as a series of horizontal lines, one on top of another. (This is similar, but not identical, to the way in which we drew polygons as a series of vertical lines in the last chapter.) We can scale the image along its horizontal axis by adding and subtracting horizontal lines. If the image is to become larger, we'll repeat some of the lines two or more times. If the image is to become smaller, we'll subtract some of the lines. We'll use an error term to determine if and when lines need to be added or subtracted. Before drawing a pixel, we'll add the value of *percent* to this error term and compare the sum to 100. If the error term is equal to or greater than 100, then we'll draw the next row of pixels and subtract 100 from the error term. We'll keep doing this until the error term is less than 100.

It's not hard to see how this works. If the value of *percent* is 50 — that is, if we are reducing the image to 50 percent of its original size — *percent* will need to be added to the error term twice before a row of pixels is drawn. This will cause the drawing routine to skip every other row of pixels, reducing the image to 50 percent of its original size in the vertical dimension. (We'll be doing the same thing in the horizontal dimension, as you'll see in a moment.) If percent is equal to 200, we'll need to subtract 100 from the error term twice before its value becomes less than 100, so every row will be drawn twice. Values between these extremes will scale the image to 20 percent, or 75 percent, or 150 percent, or whatever percent of its original size.

We'll call this error term *yerror,* because it will be used for scaling on the y axis:

```
int yerror=0;
```

Earlier, we established the x position at which we'll draw the scaled image, but not the y position. That's because we want to adjust the y position as the image is scaled. When the image is small, we'll put it near the vertical center of the display. As it increases in size, we'll move it up the screen so that the bottom of the image won't run past the end of the screen buffer. We can use the *percent* variable to calculate a reasonable y position and store it in the integer variable *screeny:*

```
int screeny=(200-percent)/2;
```

Before we can begin drawing, we need to know the offset in video memory at which the pixel in the upper-left corner of the scaled image is to be drawn. The coordinates of that corner are stored in the constant *SCREENX* and the variable *screeny.* From these, we can calculate the offset using the standard formula and store it in the unsigned integer variable *scrptr:*

```
unsigned int scrptr=screeny*320+SCREENX;
```

We also need to know the offset into the *bitmap* array of the next pixel in the original image. This pixel may or may not actually be drawn, depending on the current scaling percentage. We'll store the offset in the unsigned int variable *bitptr:*

```
unsigned int bitptr=0;
```

Because we'll be drawing the bitmap as a series of horizontal lines, we'll use a *for()* loop to loop through all the lines in the bitmap:

```
for (int y=0; y<IMAGE_HEIGHT; y++) {
```

Should we draw the next row? Or should we skip over it to reduce the size of the bitmap? That depends on the value of *yerror*. We'll add the percentage value to *yerror,* then check to see if it's greater than 100:

```
yerror+=percent;
while (yerror>100) {
  yerror-=100;
```

If the error term is equal to or greater than 100, we'll subtract 100 from it and start drawing the row. We use a *while()* statement here rather than an *if(),* so that the loop will be executed more than once if *yerror* is still greater than 100 after 100 has been subtracted from it. This will only happen if *percent* is greater than 100 and will cause the pixel to be drawn more than once in succession. This will make the pixel appear wider than it normally would, increasing the vertical size of the drawing (which is appropriate when the percentage is greater than 100).

Before we can begin drawing pixels, there are a few details to take care of. The first is to record the value of *scrptr* (which, as you recall, contains the current pixel offset in the screen buffer) in a variable called *old_scrptr*. We'll use this value later to calculate the offset for the next row of pixels to be drawn, as you'll see in a moment:

```
unsigned int old_scrptr=scrptr;
```

We'll also need to save the value of *bitptr*, which contains the current offset in the pixel buffer, in a variable called *old_bitptr:*

```
unsigned int old_bitptr=bitptr;
```

The Horizontal Error Term

Just as we used an error term to determine which pixels in a row are to be repeated and which are to be skipped, we'll also use an error term to determine

which *columns* of pixels are to be repeated and which are to be skipped. Because it will be used to scale the image on its x axis, we'll call this integer variable *xerror:*

```
int xerror=0;
```

We'll use a *for()* loop to iterate through all of the pixels in the row:

```
for (int x=0; x<IMAGE_WIDTH; x++) {
```

On each iteration, we'll add 100 to *xerror* to determine if we should draw the next pixel. If *xerror* is greater than 100, we'll draw the pixel and subtract 100 from *xerror* — and we'll continue drawing the pixel and subtracting 100 until *xerror* is less than 100:

```
xerror+=percent;
while (xerror>100) {
  xerror-=100;
```

Drawing a Pixel

After one heck of a lot of preparation, we are finally ready to draw the next pixel. This is a simple matter of copying the pixel value at offset *bitptr* in the *bitmap* array to the position in the screen buffer pointed to by *scrptr:*

```
screen_buffer[scrptr]=mona.image[bitptr];
```

That being done, we can increment *scrptr* to point to the position at which the next pixel will be drawn and end the innermost *while()* loop, the one that repeats the pixel drawing if the scaling percentage is greater than 100:

```
  scrptr++;
}
```

We deliberately do not increment *bitptr* to point at the next pixel in the *bitmap* buffer, because we may need to draw the current pixel more than once. As soon as the innermost *while()* loop stops executing, however, we move *bitptr* forward by one and terminate the *for()* loop that draws the row of pixels:

```
  bitptr++;
}
```

The current row of pixels has now been drawn. We'll advance *scrptr* to the next row of pixels on the display. The easiest way to do this is to take the offset of the first pixel in the row just finished and add 320 (the width of the display) to it. Fortunately, we stored the offset of the first pixel of the row in the variable *old_scrptr:*

```
scrptr=old_scrptr+320;
```

If the current row of pixels in the *bitmap* buffer is being drawn more than once (that is, if the value of *percent* is greater than 100), we'll need to reset *bitptr* back to the first pixel in the row. We've stored the offset of that pixel in the variable *old_bitptr:*

```
  bitptr=old_bitptr;
}
```

And that ends the *while()* loop that draws the rows of pixels, repeating them if necessary. We now must advance *bitptr* to the next line of pixels in the bitmap and end the *for()* loop that iterates through all of the rows of the image. Advancing *bitptr* is easy, since we've just reset it to the beginning of the last row. All we need to do is add the width of the bitmap to it:

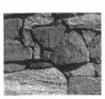
```
  bitptr+=IMAGE_WIDTH;
}
```

The scaled bitmap has now been drawn, but we'll need to move the contents of the screen buffer into video memory so that the user can see the drawing. Then we'll close all outstanding loops:

```
    blitscreen(screen_buffer);
  }
}
```

The BITSCALE.CPP Program

There's nothing left to do at this point except restore the old video mode and end the program. We've done this in so many programs now that we don't need to show the details. The complete text of BITSCALE.CPP, however, appears in Listing 6-1:



Listing 6-1 The BITSCALE.CPP program

```
// BITSCALE.CPP Version 1.0
// Scales a bitmap on the mode 13h display
//
// Written by Christopher Lampton
//   for GARDENS OF IMAGINATION (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
```

```
#include  <conio.h>
#include  <stdlib.h>
#include  <time.h>
#include  "screen.h"
#include  "targa.h"
#include  "bitmap.h"

#define   FRAMES_PER_SECOND 2

const IMAGE_X=25;
const IMAGE_Y=21;
const IMAGE_WIDTH=76;
const IMAGE_HEIGHT=96;
const SCREENX=20;
const SCREENY=0;

long ticks_ per_frame; // Number of clock ticks per
                       // frame of animation

tga_struct mona,bg;
char bitmap[IMAGE_WIDTH*IMAGE_HEIGHT];

void main()
{

  // Create pointer to video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Create offscreen video buffer:
  char far *screen_buffer=new unsigned char [64000];

  // Put display in mode 13h:
  setmode(0x13);

  // Set number of ticks per frame:
  ticks_ per_frame = CLK_TCK / FRAMES_PER_SECOND;

  // Initialize the frame timer:
  clock_t lastframe=clock();

  // Load bitmap from TGA file:
  if (loadTGA("mona.tga",&mona)) {
     setmode(oldmode);
     exit(1);
  }

  if (loadTGA("monabg.tga",&bg)) {
     setmode(oldmode);
```

continued on next page

continued from previous page

```
    exit(1);
  }

  // Set palette to TGA palette:
  setpalette(mona.color_map);

  // Keep looping until somebody hits a key:
  while (!kbhit()) {

    // Copy background image to screen buffer:
    for (unsigned int i=0; i<64000; i++)
      screen_buffer[i]=bg.image[i];

    // Expand picture from 10 to 200 percent:
    for (int percent=10; percent<=200; percent+=5) {

      // Pause until time for next frame:
      while ((clock()-lastframe)<ticks_ per_frame) ;

      // Start timing another frame:
      lastframe=clock();

      // Initialize vertical error terms:
      int yerror=0;

      int screeny=(200-percent)/2;

      // Offset of first screen pixel in row:
      unsigned int scrptr=screeny*320+SCREENX;

      // Offset of first bitmap pixel in row:
      unsigned int bitptr=0;

      // Loop through rows of bitmap:
      for (int y=0; y<IMAGE_HEIGHT; y++) {

        // Should we draw next row?
        yerror+=percent;

        // If so, repeat row until done:
        while (yerror>100) {

          // Reset error term:
          yerror-=100;

          // Save screen row offset:
          unsigned int old_scrptr=scrptr;

          // Save bitmap row offset:
          unsigned int old_bitptr=bitptr;

          // Initialize horizontal error term:
          int xerror=0;
```

```
          // Loop through pixels in row:
          for (int x=0; x<IMAGE_WIDTH; x++) {

            // Should we draw next pixel?
            xerror+=percent;

            // If so, loop through all rows to be drawn:
            while (xerror>100) {

              // Reset error term:
              xerror-=100;

              // Copy next pixel from bitmap to screen:
              screen_buffer[scrptr]=mona.image[bitptr];

              // Point to next pixel on screen:
              scrptr++;
            }
            // Point to next pixel in bitmap:
            bitptr++;
          }

          // Point to next row on screen:
          scrptr=old_scrptr+320;

          // Repeat same row in bitmap:
          bitptr=old_bitptr;
        }

        // Point to next row in bitmap:
        bitptr+=IMAGE_WIDTH;
      }

      // Move screen buffer to screen:
      blitscreen(screen_buffer);
    }
  }

  while (!kbhit());

  // Release memory
  delete bg.image;
  delete bg.color_map;
  if (bg.ID)  delete tga[ii].ID;
  delete mona.image;
  delete mona.color_map;
  if (mona.ID) delete mona.ID;
  screen_buffer;

  // Restore old video mode:
  setmode(oldmode);
}
```

Running the Program

To see BITSCALE.CPP in action, go to the directory called TEXTURE and type BITSCALE. You'll see Mona go from small to large, as in Figures 6-2a through c, then back to small again. To stop the process, hit any key, though the program won't acknowledge your request until it has run through a complete scaling cycle.

Texture Mapping

Now that we've seen how to scale an image, we're ready to develop some more versatile texture-mapping code. Surprisingly, the complete texture-mapping function will use the same principles of bitmap scaling that we just used to create BITSCALE.CPP to perform both the scaling and the rotating of polygons. By limiting ourselves to a subset of all polygons, the way we did in the polygon-drawing function in the last chapter, we will be able to reduce texture mapping to a simple scaling system, where the polygon is drawn as a series of vertical lines, each of which can be scaled individually.

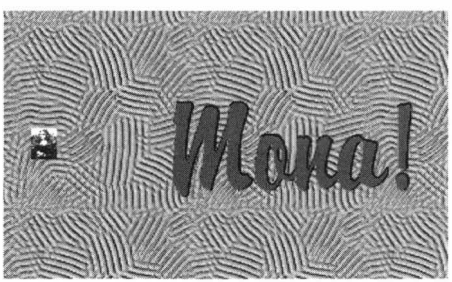


Figure 6-2a Growing...

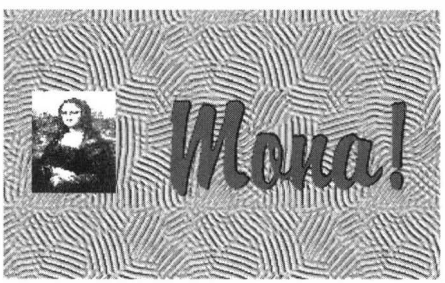


Figure 6-2b Growing...

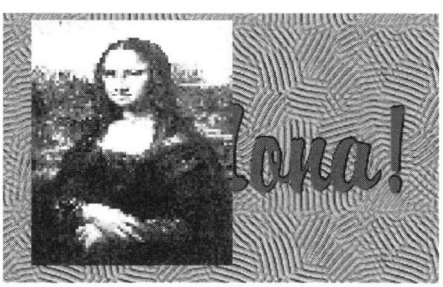


Figure 6-2c Grown!

The Ground Rules

Let's set out some ground rules for the type of texture mapping that we'll be doing in this chapter. First, we are only going to work with rectangular bitmaps. This may seem like an obvious statement, since bitmaps are almost always rectangular, but in texture mapping there are some advantages to working with bitmaps that have nonrectangular shapes, especially if the polygon onto which the bitmap is being mapped has the same nonrectangular shape. We're not going to worry about such special cases in this book, however.

Second, the polygons onto which we will be mapping these bitmaps will always have four vertices, as the polygons we generated in the last chapter did. And the left and right edges of those polygons will always be vertical. Further, each corner of the bitmap to be mapped onto the polygon will correspond to an equivalent corner of the polygon. Thus the upper-left corner of the bitmap will correspond to the upper-left corner of the polygon, and so forth. Figure 6-3 shows the relationship between a rectangular bitmap and a four-sided polygon.

The surprising thing about mapping bitmaps to this limited subset of polygons is that we don't need to know anything about the way in which the polygon has been rotated in order to map the texture onto it. All we need to know are the x,y coordinates of the four corners of the polygon. (In fact, we don't even need to know the x coordinates of the two lower corners of the polygon, just as we didn't need to know these coordinates when we were drawing filled polygons in the last chapter.) Once we know those coordinates, we can draw the polygon as a sequence of vertical lines, as in chapter 5. But instead of drawing each line in a solid color, we will copy the bitmap pixels from the equivalent

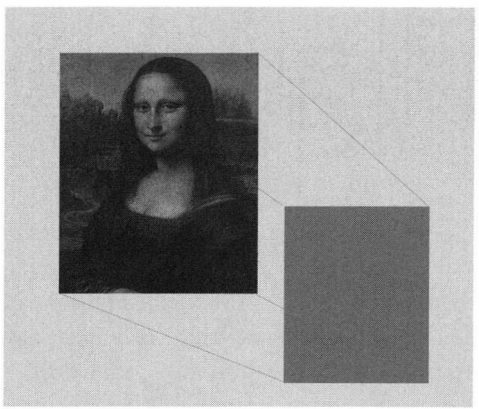


Figure 6-3 Each of the corners of the bitmap will correspond to one of the corners of the polygon

vertical line in the texture map, skipping or repeating pixels as necessary to match the length of the vertical line of pixels in the texture map to the length of the vertical line in the polygon. In effect, we will be performing a separate scaling operation on each vertical line copied from the bitmap to the polygon. (We'll also be skipping or repeating entire vertical lines, as necessary to scale the texture horizontally.) Not surprisingly, we'll use a pair of error terms to determine the precise amount by which the texture map needs to be scaled in the horizontal and vertical dimensions.

The Texture-Mapped Polygon Function

To draw texture-mapped polygons, we'll write a function called *polytext(),* which will be similar to the *polydraw()* function that we developed in the last chapter. This function will be placed in the file POLYTEXT.CPP. The prototype for the function, which will be placed in the file POLYTEXT.H, looks like this:

```
void polytext(tpolytype *tpoly,int leftx,int lefty,int rightx,
              int righty,char *screen)
```

You'll note that these parameters are similar to the ones passed to the *polydraw()* function in the last chapter. The integer variables *leftx, lefty, rightx,* and *righty* define the x,y coordinates of the upper-left and lower-right corners of the viewport that the polygon is to be clipped against. There's no *color* parameter passed to the function, because we won't be filling the polygon with a solid color. The description of the polygon itself is passed to the function in a structure of *tpolytype.* Similar to the *polytype* structure that we used in the last chapter, *tpolytype* is defined in POLYTEXT.H like this:

```
struct tpolytype {
     int x[4];
     int y[4];
     int wtext;
     int htext;
     char *textmap;
};
```

The *x* and *y* arrays work just like they did in the *polytype* structure, holding the x and y coordinates of the four corners of the polygon, starting at the upper-left corner and proceeding counterclockwise around the shape. The *wtext* and *htext* fields contain the width and height, respectively, of the rectangular bitmap to be mapped onto the polygon. The pointer variable *textmap* points at the array of type *char* that contains the bitmap to be mapped onto the polygon.

The actual polygon drawing is handled in much the same way as it was in *polydraw().* However, there is additional code here to handle the texture mapping.

Since we've already covered the polygon-drawing algorithm in chapter 5, we'll skip over that part of the function and discuss the texture-mapping additions.

At the beginning of the function, we create a pointer into the texture map array, which will initially be set to 0:

```
unsigned int tptr=0;
```

We then check to see if the polygon crosses the left edge of the viewport, as in Figure 6-4. If it does, we clip it. However, in addition to clipping the polygon as we did in chapter 5, we must also clip the texture — that is, we must reposition the *tptr* variable so that it points not at the first pixel in the texture map, but at the first pixel that falls within the viewport. To find the new position for *tptr,* we calculate three integer values: *wtemp* (which contains the width of the unclipped polygon), *offtemp* (which contains the width of the portion of the polygon that extends beyond the left edge of the viewport), and *ratio* (which represents the ratio of the width of the polygon to the width of the texture to be mapped onto the polygon):

```
int wtemp=tpoly->x[1]-tpoly->x[0]+1;
int offtemp=leftx-tpoly->x[0];
float xratio=(float)tpoly->wtext/wtemp;
```

What are we going to do with these values? We need to calculate how much of the texture map will extend beyond the left edge of the viewport. Unfortunately, this isn't necessarily the same as the portion of the polygon that extends beyond the left edge of the viewport. We'll be scaling the bitmap to fit the polygon, so one pixel in the bitmap may be equivalent to several pixels in the polygon — or vice versa. The *ratio* variable tells us what the actual scaling ratio will be. If it's

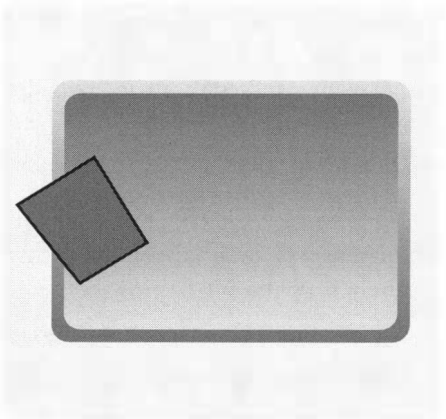


Figure 6-4 A polygon overlapping the left edge of the viewport

less than 1, then we will be increasing the size of the texture map to fit the polygon. If it's more than 1, then we will be reducing the size of the texture map.

For now, though, we'll just multiply *ratio* by the portion of the polygon that extends off the lefthand side of the viewport to get the number of bitmap pixels that would have been mapped into that portion had we been drawing it. We can then advance *tptr* past this portion of the bitmap:

```
tptr=offtemp*xratio;
```

We must also subtract this value from the *wtext* field, which contains the width of the texture map, to get the clipped width of the texture map:

```
tpoly->wtext-=tptr;
```

We then proceed to clip the polygon itself, just as we did in the *polydraw()* function. Before we can begin the actual drawing, though, we'll put the height and width of the texture map into a pair of integer variables, so that we don't need to continually reference the *tpoly* structure to get those values. (This may or may not speed up the program slightly, depending on how intelligently the compiler translates our high-level instructions into optimized machine code.)

```
int theight=tpoly->htext;
int twidth=tpoly->wtext;
```

We also need to create a pair of error terms that will tell us when it's time to advance vertically and horizontally in the texture map. First we'll create the horizontal error term, which we'll call *txerror:*

```
int txerror=0;
```

Next, we'll initiate a *for()* loop that will step across the entire width of the polygon, just as in *polydraw()*. However, at the head of this loop, we'll create a variable called *t* that will serve as a pointer into the texture map. (The *tptr* variable also serves this function, but we'll need to preserve the current value of *tptr* for later use.)

```
int t=tptr;
```

As before, we'll clip the polygon against the top edge of the viewport. This time, however, we'll also need to clip the texture. First we'll calculate the ratio of the height of the bitmap to the height of the polygon:

```
float yratio=(float)tpoly->htext/height;
```

Then we'll multiply this ratio by the portion of the polygon that extends above the top edge to calculate how much of the bitmap extends over that edge:

```
t+=(lefty-y)*yratio;
```

After the clipping of the polygon and texture map are complete, we'll establish yet another error term to help us scale the texture in the y dimension:

```
int tyerror=0;
```

Now we begin the *for()* loop that draws the vertical column of pixels, just as we did in *polydraw()*. However, instead of setting each pixel in the column to a single color, we copy a pixel from the texture map into video buffer:

```
screen[ptr]=tpoly->textmap[t];
```

Should we now advance to the next pixel in the texture map? That depends on the value of *tyerror:*

```
while (tyerror<theight) {
```

If the value of *yerror* is currently less than the height of the texture map, we add the height of the pixel column to it:

```
tyerror+=height;
```

and advance the texture map pointer to the next line:

```
     t+=twidth;
}
```

We continue advancing the pointer as long as the value of *tyerror* is less than *theight*. When the texture map needs to be reduced in size, this *while()* loop will execute more than once, skipping the pointer over the pixels that need to be omitted. If the texture map needs to be expanded, this *while()* loop will only execute on some passes through the drawing loop, causing pixels to be drawn more than once. When the value of of *tyerror* becomes greater than *theight,* the loop terminates and we subtract *theight* from the error term so that the process can begin once more:

```
tyerror-=theight;
```

Once the vertical column of pixels has been drawn, we must decide whether to advance to the next column, skip a column, or repeat the present column. The error term *txerror* helps us make this decision:

```
while (txerror<twidth) {
```

If *txerror* is less than the width of the texture map, we add the width of the polygon to *txerror* and advance *tptr* to point to the next column of pixels:

```
     txerror+=width;
     tptr++;
}
```

We continue doing this until the value of *txerror* is greater than *twidth*. This determines whether the column of pixels is skipped or repeated. Once *txerror* is greater than *twidth*, we subtract *twidth* from it and close the loop:

```
txerror-=twidth;
```

The polytext() Function

A listing of the complete *polytext()* function appears in Listing 6-2.

Listing 6-2 The polytext() function

```
#include <stdio.h>
#include <math.h>
#include "polytext.h"

// Function to draw a texture mapped polygon defined by
// polytype parameter POLY clipped against a viewport with
// upper lefthand coordinates of LEFTX,LEFTY and lower
// righthand coordinates of RIGHTX,RIGHTY.

void polytext(tpolytype *tpoly,int leftx,int lefty,int rightx,
                int righty,char *screen)
{
  float tslope,bslope;
  int vwidth,vy,vheight;

  // Initialize pointer into texture map array:
  unsigned int tptr=0;

  // If polygon crosses left side of viewport, clip it:
  if (tpoly->x[0]<leftx) {

    // Calculate offset of clip in texture map array:
    int wtemp=tpoly->x[1]-tpoly->x[0]+1;
    int offtemp=leftx-tpoly->x[0];
    float xratio=(float)tpoly->wtext/wtemp;
    tptr=offtemp*xratio;
    tpoly->wtext-=tptr;

    // Calculate slopes for top and bottom edges:
    tslope=(float)(tpoly->y[1]-tpoly->y[0])/(tpoly->x[1]-tpoly->x[0]);
    bslope=(float)(tpoly->y[3]-tpoly->y[2])/(tpoly->x[0]-tpoly->x[1]);

    // Find new endpoints for clipped lines:
    tpoly->y[0]=tpoly->y[0]+tslope*(leftx-tpoly->x[0]);
```

```
  tpoly->y[3]=tpoly->y[3]+bslope*(leftx-tpoly->x[0]);
  tpoly->x[0]=leftx;
}

// Initialize x,y coordinates for polygon drawing:
int x=tpoly->x[0];
int y=tpoly->y[0];

// Calculate the change in y coordinates for top
// and bottom edges:
int topdiff=tpoly->y[1]-tpoly->y[0];
int botdiff=tpoly->y[2]-tpoly->y[3];

// Initialize height and width of clipped polygon:
int height=tpoly->y[3]-tpoly->y[0];
int width=tpoly->x[1]-tpoly->x[0]+1;

// Clip polygon width against right side of viewport:
if (tpoly->x[1]>rightx)
  vwidth=width-(tpoly->x[1]-rightx);
else vwidth=width;

// Initialize height and width of texture map:
int theight=tpoly->htext;
int twidth=tpoly->wtext;

// Initialize top and bottom error terms:
int toperror=0;
int boterror=0;

// Initialize horizontal error term for texture map:
int txerror=0;

// Loop across width of polygon:
for (int w=0; w<vwidth; w++) {

  // Initialize temporary offset into texture map:
  int t=tptr;

  // If top of current vertical line is outside of
  // viewport, clip it:
  if (y<lefty) {
    vy=lefty;
    vheight=height-(lefty-y);
    float yratio=(float)tpoly->htext/height;
    t+=(lefty-y)*yratio;
  }
  else {
    vy=y;
    vheight=height;
  }
```

continued on next page

continued from previous page

```
    if ((vy+vheight)>righty)
      vheight=righty-vy;

    // Point to video memory offset for top of line:
    unsigned int ptr=vy*320+x;

    // Initialize vertical error term for texture map:
    int tyerror=0;

    // Loop through the pixels in the current vertical
    // line, advancing PTR to the next row of pixels after
    // each pixel is drawn.
    for(int h=0; h<vheight; h++) {
      screen[ptr]=tpoly->textmap[t];

    // Is it time to move to next row of pixels in map?
    while (tyerror<theight) {
      tyerror+=height;
      t+=twidth;
    }
    tyerror-=theight;
    ptr+=320;
  }
  x++;

    // Advance x coordinate to next vertical line:
    toperror+=abs(topdiff);

      // If so move it up...
    while (toperror>=width) {

      // If so move it up...
      toperror-=width;
      if (topdiff>0) {
        y++;
        --height;
      }
      else {
        --y;
        height++;
      }
    }
    boterror+=abs(botdiff);
    while (boterror>=width) {
      boterror-=width;
      if (botdiff>0) height++;
      else --height;
    }

//  Is it time to move to next pixel column in map?
    while (txerror<twidth) {
      txerror+=width;
      tptr++;
```

```
    }
    txerror-=twidth;
  }
}
```

A Textured Polygon Demo

To test this function, we'll rewrite the POLYDEMO.CPP program from the last chapter as TEXTDEMO.CPP. The changes will be minor. We'll create a TGA structure to load a bitmapped image into:

```
pcx_struct wall;
```

What bitmap will we use? Let's try using one of the bitmapped images from our earlier bitmapped maze program. Specifically, we'll use a portion of the image from FRONT1.TGA:

```
if (loadTGA("front1.tga",&wall)) exit(1);
```

Now let's allot some memory in the *tpoly* structure to store the portion of the bitmap that we'll be "grabbing" out of FRONT1.TGA:

```
tpoly.textmap=new char[95*79];
```

What portion of the bitmap should we grab? Let's go for the portion that shows a complete wall segment:

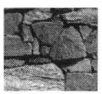
```
grab(47,20,95,79,pcx.image,tpoly.textmap);
```

The rest of the code is identical to that in POLYDEMO.CPP, except that when we set up the *tpoly* structure to be passed to the *textdemo()* function, we must include values for the width and height of the bitmap:

```
tpoly.wtext=95;
tpoly.htext=79;
polytext(&tpoly,60,40,95,175,screen);
```

The TEXTDEMO Program

The complete text of the TEXTDEMO.CPP program appears in Listing 6-3.



Listing 6-3 The TEXTDEMO.CPP program

```
#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
```

continued on next page

continued from previous page

```
#include  "screen.h"
#include  "polytext.h"
#include  "bresnham.h"
#include  "pcx.h"
#include  "bitmap.h"

void drawbox(int leftx,int lefty,int rightx,int righty,
                     char *screen);

pcx_struct wall;
char far bitmap[95*79];

void main()
{
  // Declare polygon structure:
  tpolytype tpoly;

  // Load wall bitmap:
  if (loadTGA("front1.tga",&wall)) exit(1);

  // Allot memory for wall bitmap:
  tpoly.textmap=new char[95*79];

  // Grab wall image in array:
  grab(0,0,95,79,pcx.image,tpoly.textmap);

  // Create pointer to video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Put display in mode 13h:
  setmode(0x13);

  // Set palette to ICEMAP palette:
  setpalette(pcx.palette);

  // Clear display:
  cls(screen);

  // Draw box on display:
  drawbox(60,40,95,175,screen);

  tpoly.x[0]=50;
  tpoly.y[0]=50;
  tpoly.x[1]=100;
  tpoly.y[1]=10;
  tpoly.x[2]=100;
  tpoly.y[2]=190;
  tpoly.x[3]=50;
  tpoly.y[3]=150;
  tpoly.wtext=95;
```

```
  tpoly.htext=79;
  polytext(&tpoly,60,40,95,175,screen);

  // Hold polygon on screen until keystroke
  while (!kbhit());

  // Release memory
  delete wall.image;
  delete wall.color_map;
  if (wall.ID) delete wall.ID;
  delete tpoly.textmap;

  // Restore old video mode:
  setmode(oldmode);

}

void drawbox(int leftx,int lefty,int rightx,int righty,
                char *screen)
{
  linedraw(leftx,lefty,rightx,lefty,15,screen);
  linedraw(rightx,lefty,rightx,righty,15,screen);
  linedraw(rightx,righty,leftx,righty,15,screen);
  linedraw(leftx,righty,leftx,lefty,15,screen);
}
```

Running the Program

To run TEXTDEMO.CPP, go to the TEXTURE directory and type TEXTDEMO. A texture-mapped polygon will appear on the display (see Figure 6-5). Try changing the polygon and clipping parameters and recompiling the demo. If you're *really* brave, try loading a different bitmap and mapping it onto the polygon.

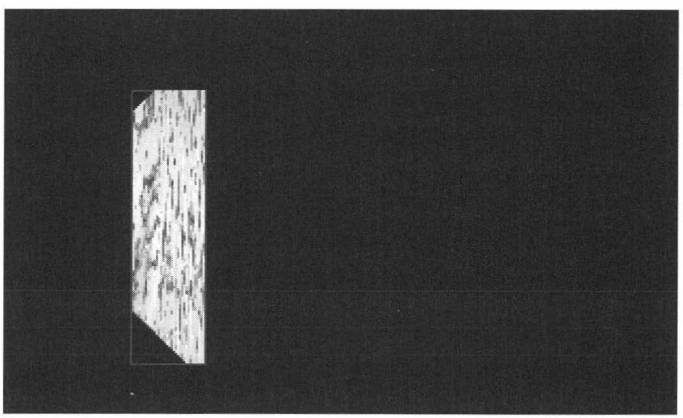


Figure 6-5 A texture-mapped polygon as drawn by TEXTDEMO.CPP

A Texture-Mapped Maze

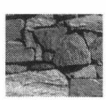
Now that we have a function to draw texture-mapped polygons, let's plug it into the POLYMAZE program from the last chapter. The main difference between the program TEXTMAZE and the earlier program POLYMAZE is that a texture map is loaded in during initialization (the same one we used in TEXTDEMO.CPP) and the *polytext()* function is called instead of *polydraw()*.

In addition, there is a new function in this program called *drawbg()*. It draws a background image behind the textured polygons. You'll recall that the maze images back in chapter 4 contained not only pictures of the walls, but of the floor and sky as well. The background image that we'll use for the *drawbg()* function will come from the file EMPTY.TGA. It is the image of an empty maze, with only floors or ceilings. We'll copy this into the viewport with the *drawbg()* function, then draw the texture-mapped polygons on top of it:

```
void drawbg(int num,char far *screen)
{
  unsigned int offset1=YWINDOW*320+XWINDOW;
  unsigned int offset2=0;
  for (int i=0; i<WHEIGHT; i++) {
    for (int j=0; j<WWIDTH; j++)
      screen[offset1++]=bg[num].image[offset2++];
    offset1+=(320-WWIDTH);
  }
}
```

The function itself is nothing more than a nested pair of *for()* loops to copy the data from the *TGA_struct* holding the floor and sky bitmap into the offscreen video buffer.

The complete text of TEXTMAZE.CPP is shown in Listing 6-4.



Listing 6-4 The TEXTMAZE.CPP program

```
// TEXTMAZE.CPP Version 1.0
// An animated tour of a 3D texture-mapped maze
//
// Written by Christopher Lampton
//   for Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  <time.h>
#include  "screen.h"
```

```
#include  "io.h"
#include  "evntmgr1.h"
#include  "bitmap.h"
#include  "pcx.h"
#include  "targa.h"
#include  "polytext.h"

#define   FRAMES_PER_SECOND 5
#define   WHICH_EVENTS KEYBOARD_EVENTS+MOUSE_EVENTS


// Function prototypes:
void display_slice(int imagenum,int leftx,int rightx,
                   char far *screen);
void drawbg(int num,char far *screen);
void drawmaze(int xshift,int yshift,int lview,int lrview,
                char *screen);

const VIEW_LIMIT = 4; // Number of squares visible both
                      // forward and to the side.
                      // (Should not be changed without
                      // adding additional bitmaps.)

const XWINDOW = 110;  // Upper left corner of view window
const YWINDOW = 0;
const WWIDTH = 188;
const WHEIGHT = 120;

// Buffers for compass picture and background:
pcx_struct compasses,pcx;
tga_struct bg[2],wall;
char far bitmap[95*79];

// Array of compass faces:
unsigned char far *compass_face[4];

// MAP OF THE MAZE
//
// Each element represents a physical square within the maze.
// A value of 0 means the square is empty, a nonzero value
// means that the square is filled with a solid cube. Map
// must be designed so that no square can be viewed over a
// greater distance than that defined by VIEW_LIMIT.

unsigned char maze[16][16]={
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
  {1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,1},
  {1,0,1,0,0,0,1,0,0,0,1,0,1,1,0,1},
  {1,0,1,1,1,1,1,0,0,0,1,0,0,0,1,1},
  {1,0,0,0,0,1,0,0,0,0,1,0,1,0,1,1},
  {1,0,0,0,0,1,0,1,1,1,1,0,0,0,1,1},
  {1,1,1,0,0,1,1,0,1,0,0,1,1,0,1,1},
  {1,0,1,0,1,1,1,0,0,0,1,0,0,0,1,1},
  {1,0,1,0,0,0,1,1,1,0,0,0,1,1,1,1},
```

continued on next page

continued from previous page

```
  {1,0,0,1,1,0,1,0,1,1,1,0,1,1,1,1},
  {1,1,0,0,0,0,1,0,1,0,0,0,0,0,1,1},
  {1,0,0,0,1,1,1,0,1,0,1,1,1,0,1,1},
  {1,0,1,1,1,0,1,0,0,0,0,1,0,0,0,1},
  {1,0,0,0,1,0,1,1,1,1,0,1,0,0,0,1},
  {1,1,0,0,0,1,1,0,0,0,0,1,0,0,0,1},
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
};

typedef struct xy {
  int x,y;
};

// Directional increments for north,east,south,west
// respectively:
struct xy forward[4]={{-1,0},{0,1},{1,0},{0,-1}};
struct xy left[4]={{0,-1},{-1,0},{0,1},{1,0}};
struct xy right[4]={{0,1},{1,0},{0,-1},{-1,0}};

// Viewer's starting position:
struct xy pos={3,8};

// Structure to hold polygon data:
tpolytype tpoly;

// Horizontal screen positions corresponding to
// visible intersections on the maze grid:
int matrix[VIEW_LIMIT+2][VIEW_LIMIT*2+2]={
  {187,187,187,187,187,  0,  0,  0,  0,  0},
  {187,187,187,238,143, 47,-49,  0,  0,  0},
  {187,187,187,179,123, 66, 10,  0,  0,  0},
  {187,187,187,158,117, 73, 31,  0,  0,  0},
  {187,187,177,144,111, 78, 45, 12,  0,  0},
  {187,185,159,133,107, 82, 56, 30,  4,  0}
};
int top[VIEW_LIMIT+2]={-19,20,36,42,46,49};
int bottom[VIEW_LIMIT+2]={138,99,83,77,73,70};

int dir=1;  // Viewer's current heading in maze,
            // where 0=north,1=east,2=south,3=west

long ticks_ per_frame; // Number of clock ticks per
                       // frame of animation

void main(void)
{
  event_struct events;
  struct xy newpos;

  // Create pointer to video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Create offscreen video buffer:
  char far *screen_buffer=new unsigned char [64000];
```

```
// Load compass picture:
if (loadPCX("compass.pcx",&compasses)) exit(1);

// Load compass faces into array:
for (int num=0; num<4; num++) {
  compass_face[num]=new unsigned char[42*41];
  grab(17+num*50,18,42,41,320,compasses.image,compass_face[num]);
}

// Dispose of compass picture:
delete compasses.image;

// Load background image:
if (loadTGA("empty.tga",&bg[0])) exit(1);
if (loadTGA("empty.tga",&bg[1])) exit(1);

// Load wall bitmap:
if (loadTGA("front1.tga",&wall)) exit(1);

// Allot memory for wall bitmap:
tpoly.textmap=new char[95*79];

// Grab wall image in array:
grab(47,20,96,79,188,wall.image,tpoly.textmap);

// Initialize event manager:
init_events();

// Calibrate the user's joystick:
if (WHICH_EVENTS & JOYSTICK_EVENTS) {
  printf("\nCenter your joystick and press button ");
  printf("one.\n");
  setcenter(); // Calibrate the center position
  printf("Move your joystick to the upper lefthand ");
  printf("corner and press button one.\n");
  setmin();    // Calibrate the minimum position
  printf("Move your joystick to the lower righthand ");
  printf("corner and press button one.\n");
  setmax();    // Calibrate the maximum position
}

// Save previous video mode:
int oldmode=*(int *)MK_FP(0x40,0x49);

// Put display in mode 13h:
setmode(0x13);

// Set palette to PCX palette:
setpalette(bg[0].color_map);

// Clear display:
cls(screen_buffer);

// Set number of ticks per frame:
```

continued on next page

continued from previous page

```
ticks_ per_frame = CLK_TCK / FRAMES_PER_SECOND;

// Initialize the frame timer:
clock_t lastframe=clock();

// Make sure we get at least one frame
// into the maze:
events.quit_game=0;

// Put initial image of maze in screen buffer:
drawbg(0,screen_buffer);
drawmaze(0,0,0,WWIDTH-1,screen_buffer);

// Point to first bg buffer:
int bgnum=0;

// Let's go for a walk in the maze:
while(!events.quit_game) {

  // Draw the maze in screen buffer:
  clrwin(XWINDOW,YWINDOW,WWIDTH,WHEIGHT,screen_buffer);
  drawbg(bgnum,screen_buffer);
  drawmaze(0,0,0,WWIDTH-1,screen_buffer);

  // Move screen buffer to screen:
  putwindow(XWINDOW,YWINDOW,WWIDTH,WHEIGHT,screen_buffer);

  // Put compass next to maze window:
  blit(20,35,42,41,screen,compass_face[dir]);

  // Pause until time for next frame:
  while ((clock()-lastframe)<ticks_ per_frame) ;

  // Start timing another frame:
  lastframe=clock();

  // Move viewer according to input events:
  getevent(WHICH_EVENTS,&events);

  // Do we want to move forward?
  if (events.go_forward) {
    newpos.x=pos.x+forward[dir].x;
    newpos.y=pos.y+forward[dir].y;
    if (!maze[newpos.x][newpos.y]) {
       pos.x=newpos.x;
       pos.y=newpos.y;
    }
  }

  // ...or do we want to move backward?
  else if (events.go_back) {
    newpos.x=pos.x-forward[dir].x;
    newpos.y=pos.y-forward[dir].y;
    if (!maze[newpos.x][newpos.y]) {
```

```
        pos.x=newpos.x;
        pos.y=newpos.y;
      }
    }

    // Do we want to turn left?
    if (events.go_left) {
      --dir;
      if (dir<0) dir=3;
      bgnum++;
      if (bgnum>1) bgnum=0;
    }

    // ...or do we want to turn right?
    else if (events.go_right) {
      dir++;
      if (dir>3) dir=0;
      bgnum--;
      if (bgnum<0) bgnum=1;
    }
  }

  // Terminate event manager:
  end_events();

  // Release memory
  delete tpoly.textmap;
  delete wall.image;
  delete wall.color_map;
  if (wall.ID) delete wall.ID;
  for (int ii=0; ii<2; ii++) {
    delete bg[ii].image;
    delete bg[ii].color_map;
    if (bg[ii].ID) delete bg[ii].ID;
  }

  for (int jj=0; jj<4; delete compass_face[jj++]);
  delete screen_buffer;

  // Restore old video mode:
  setmode(oldmode);
}

void drawbg(int num,char far *screen)
{

  unsigned int offset1=YWINDOW*320+XWINDOW;
  unsigned int offset2=0;
  for (int i=0; i<WHEIGHT; i++) {
    for (int j=0; j<WWIDTH; j++)
      screen[offset1++]=bg[num].image[offset2++];
    offset1+=(320-WWIDTH);
  }
}
```

continued on next page

continued from previous page

```
void drawmaze(int xshift,int yshift,int lview,int rview,char *screen)

// Recursive function to draw left, right and center walls for the
// maze square at POS.X+XSHIFT, POS.Y+YSHIFT, as viewed from POS.X,
// POS.Y, within the screen window bordered by horizontal coordinates
// LVIEW and RVIEW on the left and right respectively. When called
// with XSHIFT and YSHIFT set to 0, draws entire maze as viewed
// from POS.X,POS.Y out to VIEW_LIMIT squares.

{
  int vleft,vright,vwidth;

  // Return if we've reached the recursive limit:
  if ((abs(xshift) > VIEW_LIMIT) || (abs(yshift) > VIEW_LIMIT))
     return;

  // Calculate coordinates of square to left:
  int sx = pos.x + (xshift+1) * left[dir].x + yshift * forward[dir].x;
  int sy = pos.y + (xshift+1) * left[dir].y + yshift * forward[dir].y;

  // Get left-right extent of left part of view:
  vleft=lview;
  vright=matrix[yshift+1][xshift+VIEW_LIMIT+1];
  if (vright>rview) vright=rview;
  vwidth=vright-vleft;

  if (vwidth>0) {

    // If the square to the left is occupied,
    // draw right side of cube:
    if (maze[sx][sy]) {

      tpoly.x[0]=matrix[yshift][xshift+VIEW_LIMIT+1]+XWINDOW;
      tpoly.y[0]=top[yshift]+YWINDOW;
      tpoly.x[1]=matrix[yshift+1][xshift+VIEW_LIMIT+1]+XWINDOW;
      tpoly.y[1]=top[yshift+1]+YWINDOW;
      tpoly.y[2]=bottom[yshift+1]+YWINDOW;
      tpoly.y[3]=bottom[yshift]+YWINDOW;
      tpoly.wtext=95;
      tpoly.htext=79;
      polytext(&tpoly,vleft+XWINDOW,YWINDOW,vright+XWINDOW,
                 YWINDOW+WHEIGHT,screen);
    }

    // Else call function recursively to draw the view from
    // next square to the left:
    else drawmaze(xshift+1,yshift,vleft,vright,screen);

  }

  // Calculate coordinates of square directly ahead:
  sx = pos.x + xshift * left[dir].x + (yshift+1) * forward[dir].x;
```

```
sy = pos.y + xshift * left[dir].y + (yshift+1) * forward[dir].y;

// Get left-right extent of center part of view:
vleft=matrix[yshift+1][xshift+VIEW_LIMIT+1];
vright=matrix[yshift+1][xshift+VIEW_LIMIT];
if (vleft<lview) vleft=lview;
if (vright>rview) vright=rview;
vwidth=vright-vleft;

if (vwidth>0) {

  // If the square directly ahead is occupied,
  // draw front of cube:
  if (maze[sx][sy]) {
    tpoly.x[0]=matrix[yshift+1][xshift+VIEW_LIMIT+1]+XWINDOW;
    tpoly.y[0]=top[yshift+1]+YWINDOW;
    tpoly.x[1]=matrix[yshift+1][xshift+VIEW_LIMIT]+XWINDOW;
    tpoly.y[1]=top[yshift+1]+YWINDOW;
    tpoly.y[2]=bottom[yshift+1]+YWINDOW;
    tpoly.y[3]=bottom[yshift+1]+YWINDOW;
    tpoly.wtext=95;
    tpoly.htext=79;
    polytext(&tpoly,vleft+XWINDOW,YWINDOW,vright+XWINDOW,
      YWINDOW+WHEIGHT,screen);
  }

  // Else call function recursively to draw the view from
  // next square forward:
  else drawmaze(xshift,yshift+1,vleft,vright,screen);
}

// Calculate coordinates of square to right:
sx = pos.x + (xshift-1) * left[dir].x + yshift * forward[dir].x;
sy = pos.y + (xshift-1) * left[dir].y + yshift * forward[dir].y;

// Get left-right extent of right part of view:
vleft=matrix[yshift+1][xshift+VIEW_LIMIT];
if (vleft<lview) vleft=lview;
vright=rview;
vwidth=vright-vleft;

if (vwidth>0) {

  // If the square to the right is occupied,
  // draw left side of cube:
  if (maze[sx][sy]) {
    tpoly.x[0]=matrix[yshift+1][xshift+VIEW_LIMIT]+XWINDOW;
    tpoly.y[0]=top[yshift+1]+YWINDOW;
    tpoly.x[1]=matrix[yshift][xshift+VIEW_LIMIT]+XWINDOW;
    tpoly.y[1]=top[yshift]+YWINDOW;
    tpoly.y[2]=bottom[yshift]+YWINDOW;
    tpoly.y[3]=bottom[yshift+1]+YWINDOW;
    tpoly.wtext=95;
```

continued on next page

continued from previous page

```
        tpoly.htext=79;
        polytext(&tpoly,vleft+XWINDOW,YWINDOW,vright+XWINDOW,
          YWINDOW+WHEIGHT,screen);
      }

      // Else call function recursively to draw the view from
      // next square to the right:
      else drawmaze(xshift-1,yshift,vleft,vright,screen);
    }
}
```

Running the Program

To run TEXTMAZE.CPP, go to the TEXTURE directory and type TEXTMAZE. As in our earlier maze animations, you can guide yourself through the maze by pressing the arrow keys or moving the mouse.

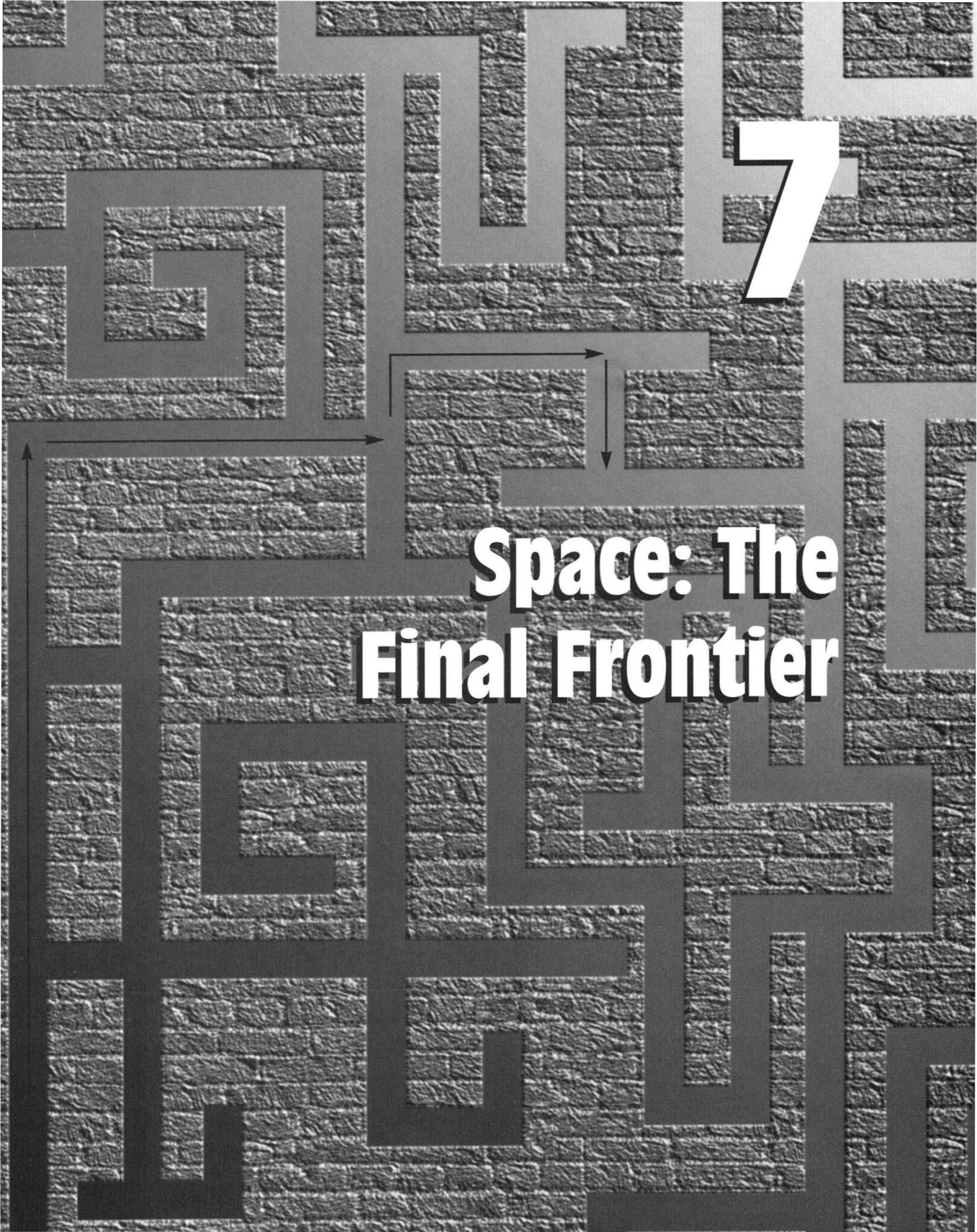
You'll notice that while the graphics in TEXTMAZE are certainly superior to the graphics in the POLYMAZE program, they are still somewhat less vivid than those in the BITMAZE program. In a sense, that's the price we pay for greatly reducing the amount of memory storage necessary to hold the bitmaps; TEXTMAZE will run in a fraction of the RAM required by BITMAZE, which means that we could greatly expand the number of bitmaps used, including bitmaps for doors, windows, other types of walls, and so forth.

And there are a number of ways in which TEXTMAZE could be improved. For instance, we could add lightsourcing to give a sense of depth to the walls. Walls at a distance from us could be drawn in a darker color than nearby walls, adding an extra sense of depth. And walls tilted at an angle to the viewer could be filled with gradient colors growing darker along the z axis of the wall. We'll talk more about lightsourcing in a later chapter, and suggest ways in which it could be added to this program.

Going Three-Dimensional

Perhaps the greatest improvement that could be made to TEXTMAZE, though, would be to turn it into a full three-dimensional game. One of the advantages of drawing polygons on the fly is that we can place the vertices of those polygons any place on the screen that we wish to; we are no longer enslaved by our predesigned bitmaps. By applying mathematical operations to the geometry of the dungeon, we can make the walls move as smoothly and realistically as we wish.

In the next chapter, we'll talk about the mathematical basis for 3D graphics. Then, in the chapters that follow, we'll introduce a new technique for generating three-dimensional dungeons that will take advantage of the techniques in the next chapter and more.


7
Space: The
Final Frontier

7

It probably doesn't come as earthshaking news to you that the video display of an IBM-compatible microcomputer is flat. Well, okay, it may be curved slightly at the edges on some models, but this minor physical quirk has no effect on the way in which graphics are programmed. We must still place pixels on the screen as if we were laying them side by side on a plane.

In mathematics a plane is an abstraction, a theoretical entity with two dimensions, which we can refer to as the horizontal and vertical dimensions, or simply as width and height. These two dimensions correspond to the two axes of a Cartesian plane, with width corresponding to the x axis and height corresponding to the y axis. Once again, this probably doesn't come as a late-breaking news item, since we've been using the x and y axes of the Cartesian plane to correspond to the width and height of the graphic images in the earlier chapters of this book.

And, in fact, this has presented something of a minor programming problem, since the image on a video display is represented by a sequence of numbers in the computer's memory that is essentially *one* dimensional. Why would we regard a computer's memory as one dimensional? Because it is a kind of line, which extends from the first address in memory to the last. This line has only a single dimension, which we can think of as a kind of length. A computer with 8 megabytes of internal memory, for instance, can be thought of as having a memory that is 8,388,608 bytes long. (A megabyte, as you're probably aware, is

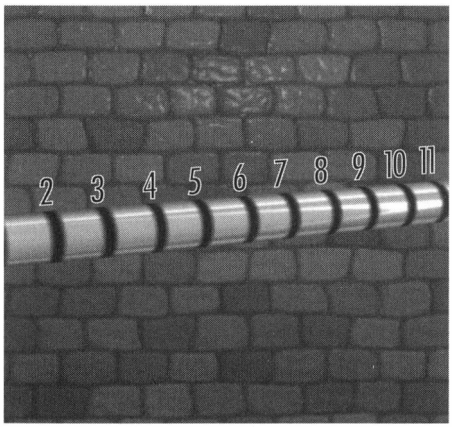


Figure 7-1 A line measured at consistent intervals with numbered increments is called a number line

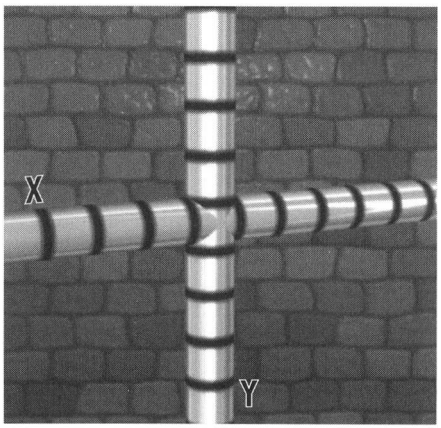


Figure 7-2 The crisscrossed axes of a Cartesian plane can be thought of as a pair of horizontal and vertical number lines

equal to 1,024 kilobytes, which are in turn equal to 1,024 bytes apiece. Thus a megabyte is 1024 x 1024, or 1,048,576, bytes.)

In math the concept of a line measured at consistent intervals with numbered increments is called a *number line*. You can see a picture of one in Figure 7-1. The memory of a microcomputer can be thought of as a kind of number line. So can the x and y axes of a Cartesian plane, as in Figure 7-2. The difference between a single number line like the computer's memory and the crisscrossed axes of a Cartesian plane is that the single number line defines only a single dimension — the length of the line itself — while the two number lines of the axes define a two-dimensional area that extends beyond the axes themselves to encompass the entire plane in which they lie.

The magic of even the simplest computer graphics is that we are making a one-dimensional number line (the memory of the computer) represent a two-dimensional plane (the flat video display of the computer). We can only get away with this because the width of the computer display is finite. If it were infinite, going on forever the way that a plane in geometry does, we couldn't pull this trick off. But the fact that the display is only 320 pixels wide (in mode 13h) allows us to use sequential chunks of our one-dimensional memory to simulate both the horizontal and vertical dimensions of space. Each 320-byte chunk of memory simulates 320 pixels' worth of horizontal space while a sequence of these chunks lined up one after another in video memory simulates the vertical dimension of space. Moving horizontally across this plane, or the 320-pixel section of it we are allowed to work with, is a simple matter of moving forward

and backward byte by byte through memory. Moving vertically is a slightly more complicated matter of moving forward and backward 320 bytes at a time through memory. Although in both cases we are moving along the same dimension of our computer's memory (since, after all, our computer's memory only *has* a single dimension), this trick of arranging memory in 320-byte chunks allows us to *simulate* movement in the two dimensions of the video display. The VGA adapter then completes the illusion by translating our one-dimensional movement through memory into *real* two-dimensional movement on the video display.

The Third Dimension

Until now, we've restricted ourselves to performing this relatively simple trick: translating the one dimension of video memory into the two dimensions of the video display. More and more, however, computer games are moving beyond the simple two-dimensional graphic effects that the video display of a microcomputer is intended to provide and into something much more complicated: the third dimension. One of the most popular categories of game software — for that matter, one of the most popular categories of *software* — is the flight simulator. The Microsoft Flight Simulator, which was up to Version 5 at the time this book was written, is one of the ten most popular general *software* titles of all time.

Flight simulators, which (as you might guess from the name) simulate the experience of flying in an airplane or other type of aircraft, use the two dimensions of the video display to represent something even more complex: the three dimensions of space.

To James Tiberius Kirk, space is the final frontier, but to a mathematician it is the three-dimensional equivalent of a plane, an abstract entity that has three dimensions. These dimensions are sometimes referred to as length, width, and breadth. Which of these corresponds to the height dimension that we referred to when we were talking about two-dimensional planes? It really doesn't matter. It could be any of these dimensions. Although when speaking of two-dimensional planes, we tend to refer to the up-and-down dimension as height and the right-and-left dimension as width, these orientations are merely incidental artifacts of the gravitational field in which we live. If you turned your computer display on its side (and didn't knock the picture tube loose in the process), you'd neatly switch width for height and vice versa, though the program that you were running would never know you'd done so.

A dimension is simply a way in which a line can be at right angles to another line. We refer to a plane as two-dimensional because we can have at most two

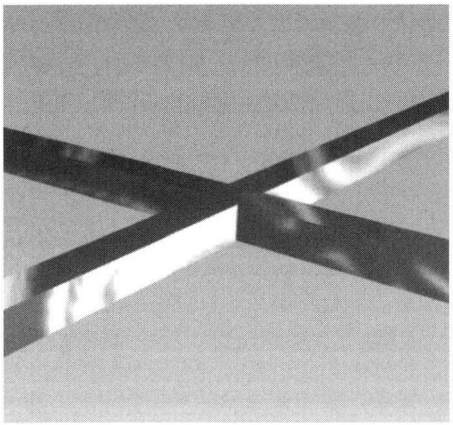


Figure 7–3 On a two-dimensional plane, we can have at most two lines at right angles to one another

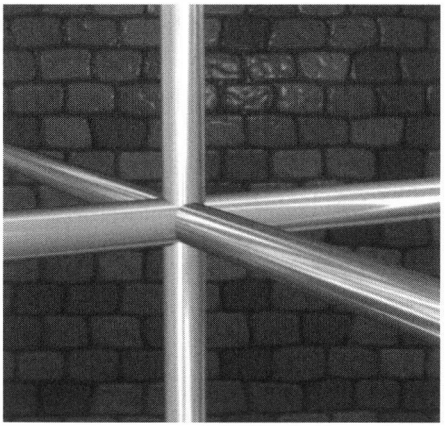


Figure 7-4 In three-dimensional space, we could have at most three lines at right angles to one another

lines at right angles to one another, as demonstrated in Figure 7-3. If we could draw lines in space, we could have up to three lines simultaneously at right angles to one another, as in Figure 7-4.

Three-Dimensional Cartesian Space

The easiest way to deal with dimensions mathematically — or even nonmathematically, if you want to cut through all of the confusion about which dimension goes up and down and which dimension goes right and left and so forth — is to think of them in terms of Cartesian axes. The Cartesian plane is called a plane because it has two axes and therefore two dimensions. It follows, then, that if we added a third dimension we could transform our two-dimensional Cartesian plane into a three-dimensional Cartesian space.

And, in fact, we can, as shown in Figure 7-5. Just as the two axes of the traditional Cartesian plane — the x and y axes — run perpendicular to one another, so the third axis of Cartesian space runs perpendicular to both of these axes. Naturally enough, we refer to it as the z axis. (The fact that mathematicians have chosen to use the letters so near the end of the alphabet to designate the Cartesian axes doesn't leave much room for us to give a name to a fourth axis, should we decide to create a graph for a four-dimensional Cartesian hyperspace. Fortunately, we won't have to do that in this book — and most of us will never have to do it in our lives.)

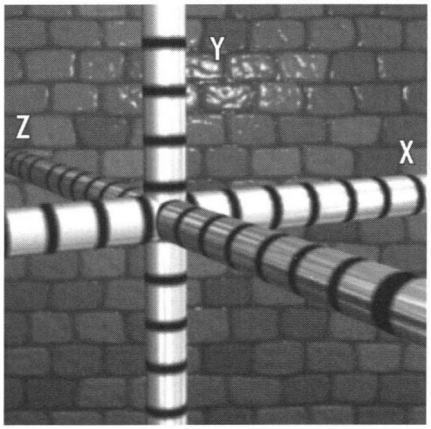


Figure 7-5 By adding a third axis, we can create a three-dimensional equivalent of a Cartesian plane

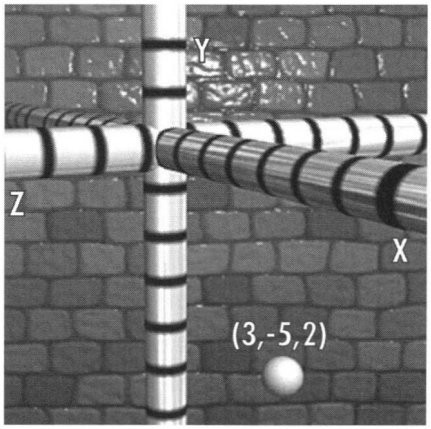


Figure 7-6 The point is at coordinate 3, -5, 2, relative to the three axes of this Cartesian space

Just as any point on a Cartesian plane can be designated by two coordinates representing its position relative to a pair of x,y axes, so any point in Cartesian space can be designated by three coordinates representing its position relative to a trio of x, y, and z axes. For instance, the levitating point in Figure 7-6 is aligned with the x, y, and z axes in the illustration at positions equivalent to the 3 on the x axis, the -5 position on the y axis, and the 2 on the z axis. Thus we can identify this point as being at position 3,-5,2 in Cartesian space — or, at least, in the particular Cartesian space represented by this picture. If we moved our axes to different positions in space without moving this point, the coordinates of the point would change. So when we say that this point has the coordinates 3,-5,2, we are actually saying it has those coordinates relative to the origin of this particular Cartesian system. As in a two-dimensional Cartesian coordinate system, the origin of a three-dimensional coordinate system is the point where all three axes come together at position 0.

The Three-Dimensional Illusion

So far, we've treated the video display of the computer as equivalent to a two-dimensional Cartesian plane, with the x axis running horizontally across the top of the display and the y axis running vertically down the left side. Is it possible also to treat the display as a three-dimensional Cartesian space? Yes and no. As we've noted, the display is essentially flat, so we don't really have a third

dimension to deal with. But we've already worked out a trick that translates the one-dimensional memory of the microcomputer's video memory into the two dimensions of the video display. So we should be able to use similar, if considerably more complex, trickery to translate the two dimensions of the video display into the illusion of a three-dimensional world.

This is exactly what almost every flight simulator on the market does. Pick up a copy of the Microsoft Flight Simulator or Spectrum Holobyte's Falcon 3.0 or Origin System's Strike Commander, and you'll see a three-dimensional world stretching away from you out the window of the aircraft simulated by the program. Of course, this world isn't *really* three-dimensional. It's actually a collection of pixels on a flat video display. But the flight simulator program produces the illusion that the world outside the window is three-dimensional, just as the masterly painters of the Renaissance produced the illusion of a third dimension in the dramatic lines of perspective that they were so fond of adding to their paintings.

There's no *real* third dimension in these images because there's no optic parallax, the effect that occurs when each of your two eyes sees the world from a slightly different angle. This effect *can* be achieved on a computer video display through the use of certain kinds of 3D glasses, such as the ones with the red and blue lenses, but most people aren't fond of donning special goggles every time they sit down to play a computer game. And besides, the three-dimensional illusion produced by a flight simulator is every bit as good as the three-dimensional illusion produced by a television show or a motion picture, both of which are depicted on flat screens. Not many people complain that television pictures look flat.

Why not? Why aren't we frustrated by the two-dimensionality of television, motion picture, and flight simulator images? Because optic parallax isn't all that there is to three-dimensional vision. In fact, it's just a small part of three-dimensional vision, an evolutionary trick picked up by our distant, tree-dwelling ancestors that made it a little less likely that they would misjudge the distance to a branch and go crashing ignominiously (and possibly fatally) to the ground.

Putting Perspective in Perspective

What really gives an image the look of three-dimensionality is perspective, the way in which objects appear to grow smaller and larger as their distance from the viewer increases and decreases, respectively. This is the result of the way in which rays of light reflected from (or produced by) those objects converge on our eyes. We'll talk about this at considerably greater length in later chapters, because it is the key to some of our most vivid and realistic maze creation programs. For now,

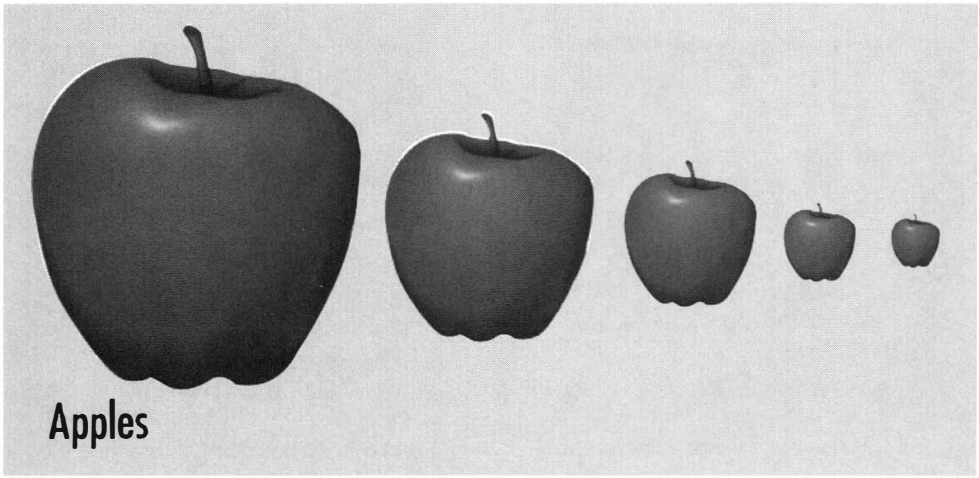


Figure 7-7 This apple appears to grow smaller the farther away from us it is

however, we can regard perspective as a simple increase or decrease in the apparent visual size of an object with distance, as in Figure 7-7.

Actually, we can be a little more specific than that. Perspective is the way in which points in our visual field tend to converge on the origin of our visual systems with distance. How's that again? Imagine that the field of view that you can see with your eyes is a kind of two-dimensional Cartesian viewing plane, ignoring for a moment the third dimension. Imagine further that at the center of

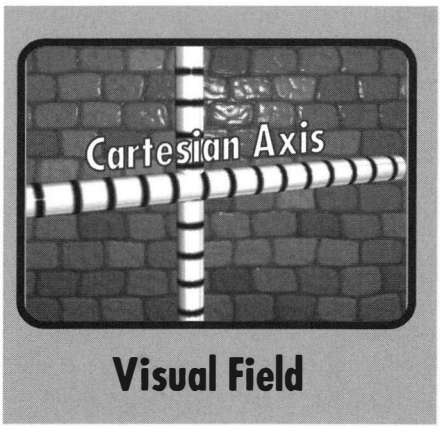


Figure 7-8 Imagine that your visual field is a two-dimensional Cartesian plane with the origin in the center

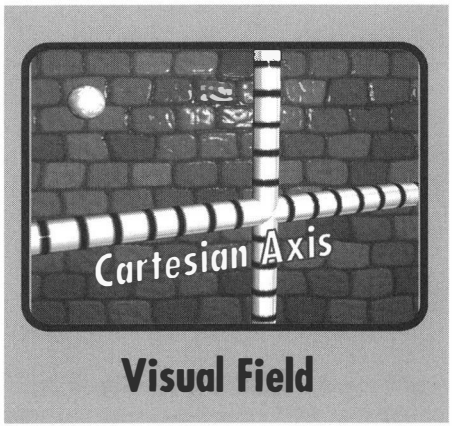


Figure 7-9 This point is the same as the one in Figure 7-10, but viewed from a greater distance

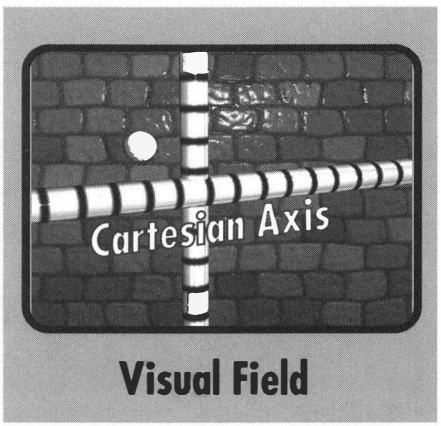


Figure 7-10 This point is the same as the one in Figure 7-9, but viewed from a closer distance

this viewing plane is the origin of a pair of x,y Cartesian axes (see Figure 7-8). Now go stand in a large open field and position yourself so that some small object — we'll imagine that this is a point in Cartesian space — is slightly off center in your visual field. Walk toward the center of your visual field (*not* toward the object) and then back away from it again. Notice that the object appears to move farther away from the center of your visual field as you move forward, and closer to the center of your visual field as you move away from it.

We can simulate this on a two-dimensional Cartesian graph by putting the origin in the center of the graph and moving a point closer to and further away from that origin to make it appear that the point is getting closer to us, then further away from us. For instance, we can imagine that the point on the graph in Figure 7-9 is the same point as the one on the graph in Figure 7-10, viewed from a greater distance. How do we know that it is viewed from a greater distance in Figure 7-9? Because it is closer to the origin.

We can imagine that the video display of a microcomputer has a z axis that runs into and out of the screen, as shown in Figure 7-11. For now, we'll regard this axis as being positive going into the display — that is, the numbers on the axis go up as we move forward into the display and go down as we move backward toward the viewer. If the origin of the z axis is right on the display itself, the negative points on the axis will be on the same side of the display as the viewer and the positive values will be on the opposite side.

This allows us to treat the display as a three-dimensional Cartesian space, at least in our imaginations. To get this three-dimensional information out of our

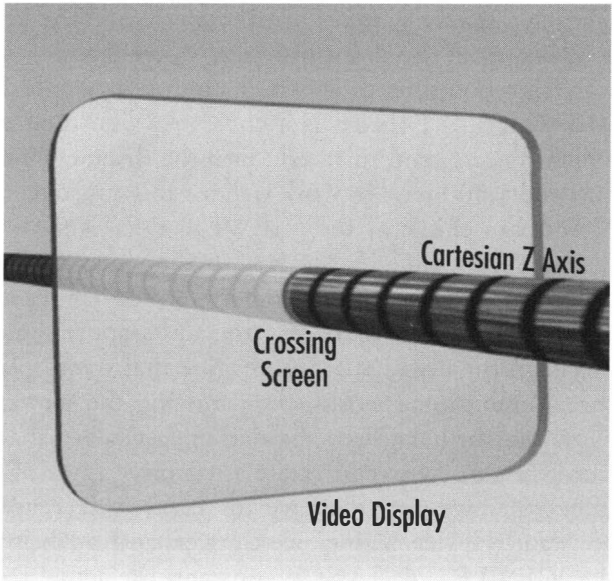


Figure 7-11 We can imagine that the video display of a microcomputer has a z axis that runs into and out of the screen

imaginations and into our programs, we can develop a simple pair of equations for adding perspective information to the position of a point on a Cartesian graph:

```
x_persp = x / z
y_persp = y / z
```

where the *x*, *y*, and *z* variables represent the x, y, and z coordinates of the point in Cartesian space and the *x_persp* and *y_persp* variables represent the perspective-adjusted projections of these points onto a two-dimensional Cartesian plane — such as a video display. How do these equations work? Basically, they add z axis information to the x and y axis information on the screen. Dividing the z axis value into the x and y axis values causes the x and y axis values to become smaller as the z value becomes larger, just as they appear to in your visual field.

Actually, the equations given above cause the x and y axis values to become smaller a bit *too* quickly for most 3D graphics systems. We need a way to slow down the process a bit, so that the points don't converge immediately on the origin as they move away from us along the imaginary z axis. One way is to multiply the x and y values by a constant, like this:

```
x_screen = viewer_distance * x_space / z_space;
y_screen = viewer_distance* y_space / z_space;
```

where *x_space*, *y_space*, and *z_space* are the three-dimensional Cartesian coordinates of the point in space; *x_screen* and *y_screen* are the two-dimensional Cartesian coordinates at which the point would be drawn on the screen; and VIEWER_DISTANCE is a constant value used to fine-tune the rapidity with which objects tend to recede into the distance. Generally, values between about 100 and 1,000 tend to work well for this constant.

You can think of the VIEWER_DISTANCE constant as representing the distance from the viewer's eyes to the video display, measured in pixels. (There aren't any pixels in the space between the viewer and the display, of course, but we can measure this space using units approximately equivalent to the size of pixels on the display.) If we imagine that the display is a window into a genuine three-dimensional world, then moving the viewer closer to the display would allow him or her to see a wider angle view of that world. Moving further away from the display would create a narrower angle of view. And this is precisely the effect that we get when we increase or decrease the VIEWER_DISTANCE constant. However, since we can't expand or shrink the display itself along with the VIEWER_DISTANCE constant, the actual result is that more visual data is crammed into the width of the viewport when the value of the constant is low and less data is crammed into the viewport when the value is high.

Two-and-a-Half Dimensions?

This is one of the tricks that flight simulators use to give the illusion of a three-dimensional world on the video display of a microcomputer. (We'll talk about some more of these tricks in a moment.) Most flight simulators use a graphics system known as polygon-fill graphics, in which objects in space are represented as a series of flat shapes with straight sides. In math a flat shape constructed from a series of straight sides, like the one in Figure 7-12, is called a polygon. A polygon depicted on the computer display with its interior filled by a solid color is called a filled polygon. Thus, polygon-fill graphics systems build up a three-dimensional "world" out of filled polygons.

The only information needed to describe a polygon in Cartesian space are the coordinates of the points at which the straight lines that make up the "sides" of the polygons come together. These points are called vertices, and can be represented like any other points in space — by a trio of Cartesian coordinates. A polygon floating in space can be represented in a computer program such as a flight simulator as a list of coordinates, going clockwise or counterclockwise around the polygon. And this is what a lot of flight simulators do.

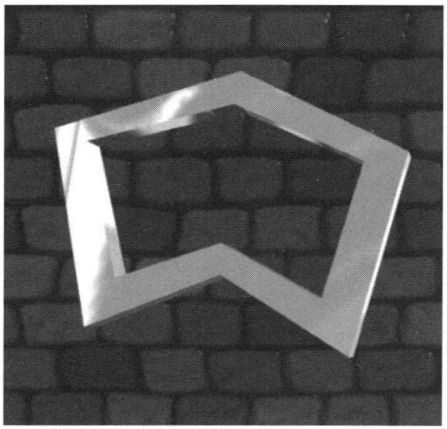


Figure 7-12 A flat shape constructed from a series of straight sides is called a polygon

Once you have a polygon represented in this form, it can be "projected" onto the display using the perspective equations that we discussed earlier, or some variation thereon. That is, the three-dimensional coordinates of the vertices within the "world" of the flight simulator — which is actually just a segment of Cartesian space, the way that the computer screen is a segment of a Cartesian plane — can be converted into two-dimensional screen coordinates by dividing the x and y coordinates (after multiplication by a constant) by the z coordinates. The polygon that connects those points on the display can then be drawn.

This is a straightforward programming problem and much of my book *Flights of Fantasy* is concerned with solving it. Most flight simulators solve it in one way or another, and some of the more recent ones have come up with very sophisticated solutions indeed. For instance, some flight simulators now employ polygon-smoothing techniques that give the ordinarily flat polygons a smooth, curved look, or texture-mapping techniques that fill the polygons with complete bitmaps rather than just solid colors. (We talked about simple forms of texture mapping in chapter 6.)

Is this what we'll have to do in order to bring a convincing three-dimensionality to our maze programs? Will we need to represent our maze as a series of fully three-dimensional polygons and perform all of the operations on these polygons that a flight simulator does, including edge clipping, hidden surface removal, and perspective projection? Not necessarily. Although we *could*

create a maze program in precisely this manner — the Ultima Underworld games from Origin, for instance, seem to go some distance in this direction — it isn't easy to use flight simulator techniques to represent a maze. For one thing, there are often a lot more polygons involved in building a maze than in building the sort of polygon world that most flight simulator craft fly around in. And the player can get a lot closer to those polygons in a maze than an airplane usually does to ground scenery, which demands greater realism in the special effects of a maze program than the typical flight simulator provides (though the realism quotient of some recent flight simulators has increased amazingly).

Fortunately, we don't have to go a full three dimensions in order to make our maze program *seem* three-dimensional. That's because a maze is essentially a two-dimensional object. The mazes that we've created in earlier chapters, for instance, were represented by two-dimensional arrays, because there really wasn't any third-dimensional information that we needed to know about in order to draw them. Although the mazes that we drew *looked* three-dimensional, the third dimension — which we can think of as the height of the maze blocks above the ground — wasn't really there. Why? Because all of the maze blocks were the same height. Thus information about the height of individual blocks — or anything else — didn't need to be stored in the array.

Indeed, if you look at most maze games on the market (the Ultima Underworld games once again excepted), you'll see that there is relatively little height information in them. In Wolfenstein 3D, for instance, all of the walls are the same height. The Wolf 3D maze is essentially two-dimensional. Yet we get the illusion that we are moving through it in three dimensions. You might say that the mazes in most games are actually *two-and-a-half-dimensional.*

In this book, although I'll make a nod toward full three-dimensional maze creation, we'll talk mostly about two-and-a-half-dimensional mazes. Why? Because two-and-a-half-dimensional maze-generation systems are the best methods currently available for generating high-speed maze games. Later, in the chapter on heightmapping, we'll talk about ways of crunching genuine three-dimensional height information into our two-and-a-half-dimensional mazes, but for the most part we'll fake the third dimension. This will allow us to produce high-speed animation, which in turn will let us place our programming resources where they really belong: on creating as realistic *appearing* an environment as possible. Don't worry; the player will never know that the third dimension really isn't there.

Nonetheless, we'll also be discussing the theory of fully three-dimensional programming, because we'll be needing it in later chapters to give the illusion that there is a third dimension present in the maze worlds we create. And the first of these three-dimensional concepts that we'll need to discuss — one that applies

equally well to programming in two-dimensional environments — is the concept of a direction vector.

Direction Vectors

A Cartesian coordinate system is a terrific way of designating the position of a point, either on a plane or in three-dimensional space (or in any number of dimensions, if you should have need of them). It can also be used to create other types of shapes, such as lines. In fact, we did just this in chapters 2 and 3, where we drew lines between points on screens using Bresenham's algorithm.

In that case we described the lines by giving the coordinates of their endpoints, like this:

```
x_start,y_start - x_end,y_end
```

where *x_start,y_start* are the coordinates of the end of the point where we began drawing the line and *x_end,y_end* are the coordinates of the point where we stopped. Indeed, this is all the information that we need to completely describe a line in terms of the Cartesian plane on which the line lies. If we need to, we can similarly describe a line in three-dimensional space by using three coordinates for each endpoint, like this:

```
x_start,y_start,z_start  -  x_end,y_end,z_end
```

There is, however, another method that can be used to describe a line in Cartesian coordinates, one that will often be more useful for our purposes than this. That method is to describe the line using its direction vector and magnitude.

The magnitude of a line is simply its length, in terms of the same unit of measurement used along the x and y axes of the coordinate system. (Since these units can correspond to any real-world units that we want them to — miles, fathoms, light-years — we'll refer to them as Cartesian units or just as units.) A direction vector is something slightly more complicated.

In mathematics the term vector is sometimes used to describe any one-dimensional array of numbers. For instance, the coordinate pair used to describe the position of a point on a two-dimensional plane (or the coordinate triple used to describe the position of a point in three-dimensional space) can be called a vector. However, the term vector is interchangeably (and sometimes confusingly) applied to a special pair (or trio) of numbers that can be used to describe the direction of a line in a Cartesian coordinate system. We'll refer to such a pair (or trio) of numbers in this book as a direction vector.

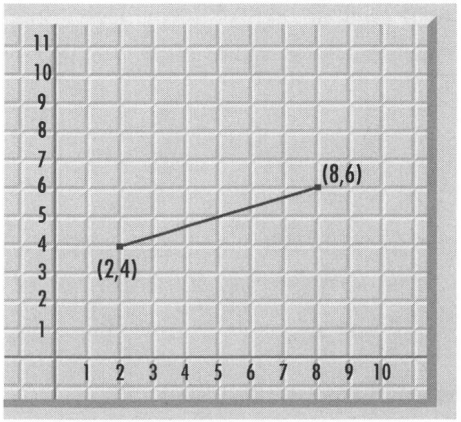


Figure 7-13 A line from coordinates 2,4 to 8,6

In a two-dimensional Cartesian coordinate system, a direction vector is a pair of numbers that tell us how much the x and y coordinates of a line change from the starting point of a line to the ending point — or, for that matter, between any two arbitrary points on the line. The first number, which represents the change in the x coordinate, is called the x component of the vector and the second number, which represents the change in the y coordinate, is called the y component of the vector. A direction vector looks an awful lot like a coordinate pair. And, in fact, we can translate easily between direction vectors and coordinate pairs. For instance, the direction vector for a line can be easily derived from its starting points and ending points, by subtracting the starting x point from the ending x point and the starting y point from the ending y point, like this:

```
x_vector = x_start - x_end;
y_vector = y_start - y_end;
```

where *x_start,y_start* are the components of the start of the line, *x_end,y_end* are the components of the end of the line, and *x_vector,y_vector* are the x and y components of the direction vector for the line. Does it matter which end of the line we start at and which end we end at? Theoretically, it does, since the x and y components of a direction vector that runs in one direction will have the opposite signs from a direction vector running in the opposite direction. For many practical purposes, though, we can ignore this. For instance, in drawing a line on the screen, it doesn't matter which end we finish drawing at, since the user doesn't usually see it being drawn. For other purposes, it will matter. In most cases, however, it will be obvious which end of the line is which.

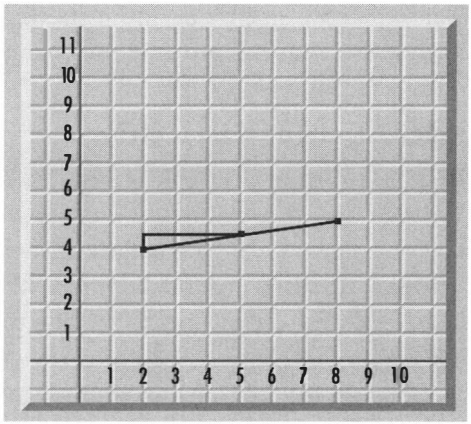


Figure 7-14 Measuring the direction vector over half the line is equivalent to measuring it over the entire line

For example, the line in Figure 7-13 runs from coordinates 2,4 to coordinates 8,6. If we treat the 2,4 end as the starting point and the 8,6 end as the ending point, the direction vector for this line would be 6,2, where 6 is 8 minus 2 and 2 is 6 minus 4. If we chose to reverse the ends of the line, starting at the 8,6 end and ending at the 2,4 end, it would reverse the vector as well to -6,-2.

A direction vector is essentially a ratio, showing the relationship between the changes in the x and y values along the length of the line. So the numbers themselves don't matter as much as the relationship between them. We can change the numbers in the vector without changing the direction that the vector represents, as long as we don't change the *ratio* between the numbers. Thus we can multiply both components of the vector by the same number or divide both components of the vector by the same number without fundamentally changing it, since this leaves the ratio between the numbers intact. For instance, the vector 6,2 of the line in Figure 7-13 can also be expressed as 12,4 (where both components have been multiplied by 2) and as 3,1 (where both components have been divided by 2).

This means that we can obtain the direction vector by measuring the change in the x,y coordinates along any segment of the line, as well as along the entire length (or magnitude) of the line. For instance, in Figure 7-14 we obtain the direction vector of the line from Figure 7-13 by measuring the change in the x,y coordinates from the starting point at 2,4 to the midpoint at 5,5. The direction vector that we get in doing so is 3,1, which we just saw in the last paragraph is equivalent to the direction vector 6,2 that we got from measuring the change in the x,y coordinates along the entire length of line. (In fact, it is the direction

vector that we got by dividing the 6,2 vector by the constant 2, not surprising considering that it represents the change in x and y along half the line.)

Vector Magnitude

Since a direction vector is a ratio rather than a pair of absolute lengths, it does not give us a length for the line the vector is describing. And, in fact, we don't always need to know the length of such a line. In many cases, as we'll see in later chapters, the direction of the line is all we need to know. (In some specific cases, such as the so-called normal vector that describes the orientation of a plane in space, the line described by the vector is indefinitely long, so talking about its length is meaningless. More about normal vectors in chapter 8.)

Nonetheless, there will obviously be times when we need to know the length of a line as well as its direction vector. (In fact, there will be times when the length of a line is the information that we really need and the direction vector for the line can be ignored altogether. This is frequently the case when we need to know the distance to a point in space in order to make perspective calculations.) Thus we need to calculate the length of a line in Cartesian units based on the coordinates of its endpoints. In some cases this will be a trivial matter, especially in the case of lines that are parallel to one of the Cartesian axes. In Figure 7-15, for instance, the line runs parallel to the x axis from coordinates 1,3 to coordinates 7,3 so we can easily calculate that this line is 6 units long.

But what about the line in Figure 7-16, which runs from starting coordinates 3,2 to ending coordinates 5,7? What is the length of this line? There are a couple of different ways in which we can calculate it. For the time being, we'll fall back

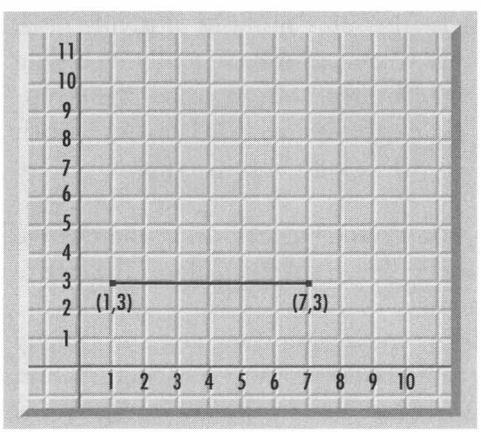


Figure 7-15 A line running parallel to the x axis from coordinates 1,3 to coordinates 7,3

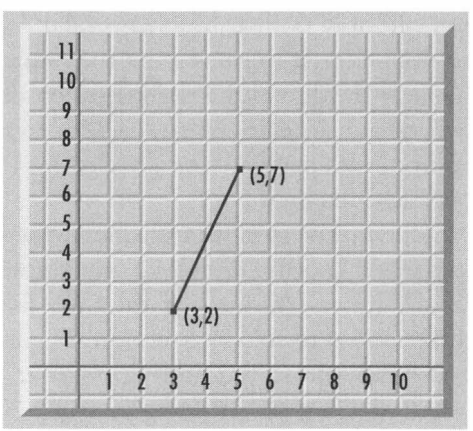


Figure 7-16 A line from coordinates 3,2 to coordinates 5,7

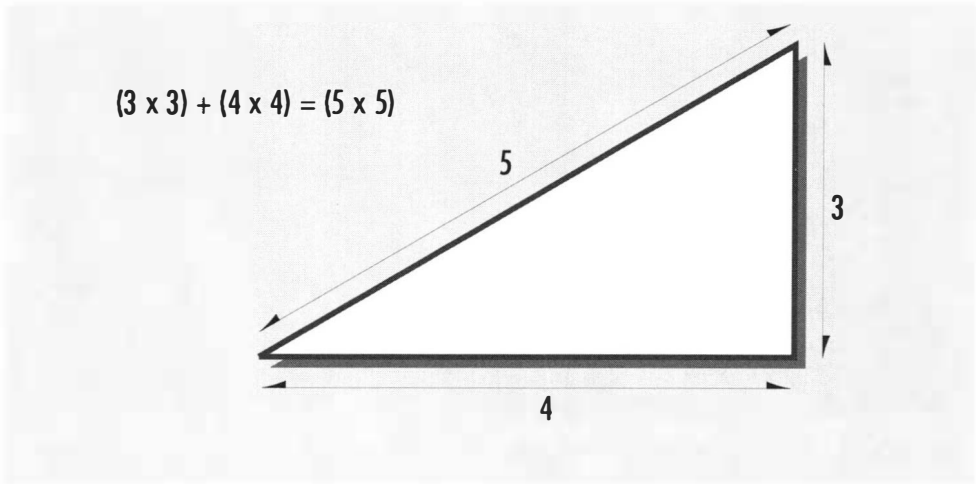


Figure 7-17 Yes, the square of the hypotenuse really *is* equal to the sum of the squares of the other two sides

on the familiar Pythagorean theorem, which states that the square of the hypotenuse of a right triangle is equal to the sums of the squares of the other two sides. In Figure 7-17, for instance, we see a right triangle in which — by golly — Pythagoras' theorem turns out to be true!

Although you may have suspected that you'd never need this theorem again after graduating from school, it comes in quite handy for calculating the length of a line within a Cartesian system, since it turns out that any line in a Cartesian coordinate system can be treated as the hypotenuse of a right triangle and the changes in the x and y coordinates between one end of this line and the next can be treated as the other two sides (see Figure 7-18).

So how do we calculate the length of the line by treating it as the hypotenuse of a right triangle? Well, we're trying to find the length of the hypotenuse, and we know the lengths of the other two sides. So what we need to do is square each of the other two sides, add them together, and take the square root of the result. That will give us the length of the line. To get the lengths of the two sides, we need to calculate the changes in the x and y coordinates by subtracting the starting x from the ending x and the starting y from the ending y, like this:

```
x_change = x start - x end;
y_change = y_start - y_end;]
```

Then, to get the distance along the hypotenuse, we must square each of these, add the sums, and take the square root, like this:

```
distance = sqrt(x_change*x_change + y_change*y_change)
```

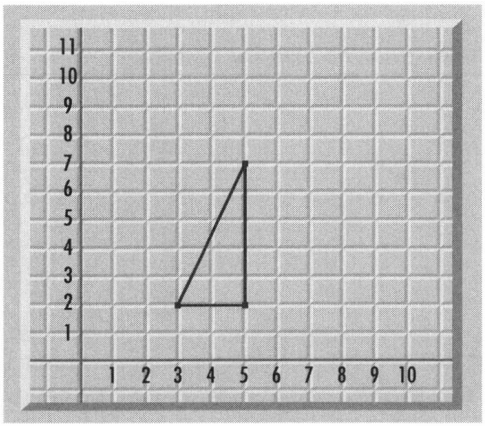


Figure 7-18 The line from Figure 7-16 as the hypotenuse of a right triangle. The changes in the x and y coordinates along the length of the line are the other two sides

where *sqrt* is a standard library function that we can call to obtain a square root. Thus we can measure the distance from one end of the line in Figure 7-16 by first calculating the change in x (which is 2) and the change in y (which is 5), then performing the necessary Pythagorean calculations:

```
distance = sqrt(2*2+5*5)
```

or

```
distance = sqrt(4+25)
```

or

```
distance = sqrt(29)
```

or

```
distance = 5.385
```

Thus the length — or magnitude — of the line in Figure 7-16 is 5.385 Cartesian units. (In the above expressions we've observed the mathematical convention that multiplication operations are performed before addition operations unless otherwise noted. Since C and C++ — as well as most other programming languages — observe this convention when evaluating expressions, we don't have to worry that the operations will accidentally be performed in a different order.)

Note that most of the concepts that we've discussed above apply to three-dimensional Cartesian space as well. A three-dimensional direction vector is the same as a two-dimensional vector, except that it has three numbers in it, representing the change in the x, y, and z coordinates along a given length of the line. Needless to say, these are referred to as the x, y, and z components of the vector.

Now that we've gotten down some basic vector concepts, we can move on to a related topic that will also be very important in the programs that we create in this book. That's the concept of polar coordinates.

Polar Coordinates

When dealing with both two-dimensional planes and three-dimensional space, it's important to be able to locate points within those planes or spaces. We've already discussed one method for doing so, the so-called Cartesian coordinate system. But there's a second method that is also commonly used, and that will sometimes be more relevant to our purposes. That's the polar coordinate system.

You've probably used the polar coordinate system on a number of occasions. In fact, when not programming graphics on the screen of a microcomputer, you may find that you use the polar system far more often than the Cartesian. For instance, when you tell somebody that in order to get where they want to go they need to drive north for 50 miles, you are telling them how to locate their destination in terms of polar coordinates.

In a polar coordinate system, all points are located relative to an arbitrary origin point along a line. Because this line has a direction and a length, or magnitude, it is sometimes called a vector. And, in fact, it can be measured exactly as we measured lines in the last section of this chapter, using a direction vector and a magnitude. However, there are other ways of denoting the direction in which this line is pointing besides a direction vector.

A compass, such as the one in Figure 7-19, is a simple example of a polar coordinate system. The center of the compass is the origin of the coordinate system and the needle is the line used to designate polar directions. With a compass, the direction of the line is usually designated as north, east, south, west, or some point between two of those directions. To locate a point using a compass, we give the current position of the compass, the direction of the line — that is, the needle — and the distance along the line at which the point can be found. (This distance can be expressed in any unit with which the listener is familiar, such as miles or feet or nanometers, though some units are probably more appropriate for identifying certain points than others.) Thus we might say

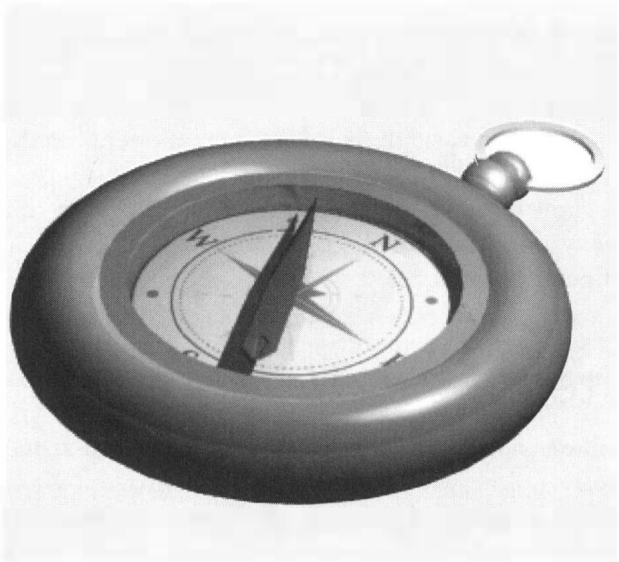


Figure 7-19 A compass is a simple example of a polar coordinate system

that a point lies 37 kilometers from our current compass location in a north-northeasterly direction.

Probably the most common way to designate the direction of a line in a polar coordinate system, though, is to measure the angle of the line relative to the origin. One of the most familiar systems for measuring angles is the 360-degree circular system. The degree was believed by the ancient astronomers to be the distance that the vector between the sun and the earth rotated every day, relative to the "fixed" stars. (If the year had been 360 days long, as the ancient astronomers believed, they would've been more or less correct, but their calculations were slightly off.) Figure 7-20 shows a 360-degree circle surrounding an origin point, with a line connecting the origin with another point to its right. The length, or *magnitude,* of the vector is 5 units. (These units belong to a completely arbitrary system that we've made up on the spot, though if we used them consistently — the way that inches, meters, and parsecs are used — they could actually serve as a satisfactory unit of measurement for vector magnitudes and other purposes.) Since we have placed the 0-degree point where it traditionally goes, straight above the origin, and have placed all the other points clockwise around the circle, we can say that the point is 10 units along a 90-degree vector from the origin.

Another system for measuring angles is the so-called *radian* system. This system is preferred by mathematicians since, unlike the degree system, it actually has something to do with the properties of a circle. You'll recall from geometry class (even if geometry class is now a distant memory) that the radius of a circle is the distance from the center of a circle to its circumference. A radian is a measure of the circumference of a circle that is equal in length to the radius. Because — here's the geometry again — the circumference of a circle is equal to two times pi (3.14...) squared, the number of radians in the full circumference of a circle is two times pi, or 6.28... (The three-dot ellipsis, you'll recall from that same math class or another you took even earlier, tells us that this number is a decimal that we are only approximating with the digits actually printed on the page. From now on, we'll generally round off such numbers and not worry about the ellipses.) Figure 7-21 shows the same point along the same vector as in Figure 7-20, but we have now translated the angle of the vector into radians. A radian is equal to 57.2948 degrees. In radians, therefore, the vector is pointed at an angle of 1.57. To translate from degrees to radians, we divide the number of degrees by 57.2948. To translate back the other way, from radians to degrees, we multiply the number of radians by 57.2948.

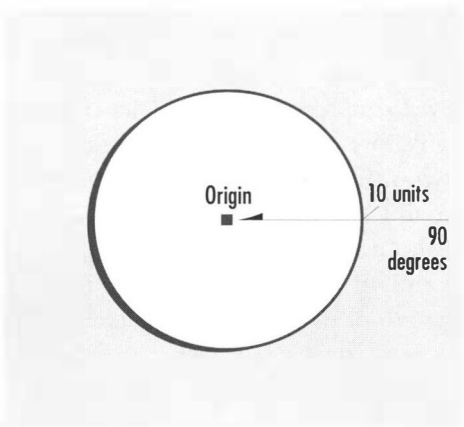


Figure 7-20 A point in a polar coordinate system is 10 units from the origin along a 90-degree vector

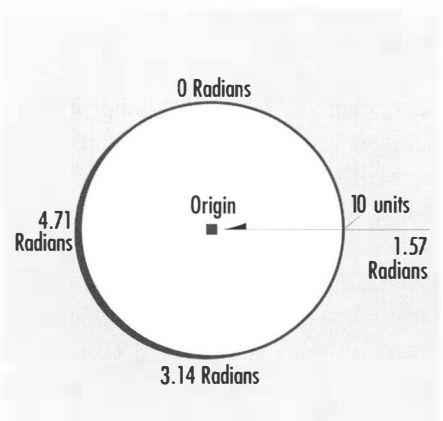


Figure 7-21 A point in a polar coordinate system 10 units from the origin along a 1.57-radian vector

Polar Direction Vectors

For polar coordinate systems to be useful to us in creating three- (or two-and-a-half-) dimensional computer games, we must be able to combine them with a Cartesian coordinate system. The easiest way to do this is to use standard Cartesian direction vectors to describe the direction between the origin and a point within a Cartesian plane. However, we'll also need to be able to describe directions in terms of angles, using either degrees or radians (or, eventually, a system that we invent for ourselves). So we'll need a method of translating back and forth between angles and direction vectors. We'll see how to do this in a moment.

Figure 7-22 shows a two-dimensional polar coordinate system superimposed over a two-dimensional Cartesian coordinate system. We've conveniently placed the origin of the polar coordinate system at the origin of the Cartesian system. This is generally the best place for it. Even when we must deal with polar coordinate systems that are not at the origin of our Cartesian coordinate system, we'll move them to the origin before performing any important mathematical operations on them, then move them back after we're done.

The position of any point within a system like this can be designated in two different ways. It can be designated according to its Cartesian coordinates, or it can be designated by the direction and magnitude of a line forming a vector between the point and the origin. For instance, the point in Figure 7-22 is at coordinates 3,0. How do we designate this as a vector? Well, the distance from the origin to the point is 3 units, so we can say that the vector from the origin to the point has a magnitude of 3. Along that length, the x coordinate of the vector changes from 0 to 3, and the y coordinate of the vector changes from 0 to 0 (which is to say that it doesn't change at all). Thus we can say that the direction of the vector is 3,0. When we've said that the point lies on a vector with a direction of 3,0 and a magnitude of 3, we've completely described its position relative to the origin, just as though we had given its Cartesian coordinates.

Similarly, the point in Figure 7-23 lies at coordinates 7,5. We've drawn a line from the origin to this point. The direction and length of this line constitute a vector from the origin to the point and can be used to describe the location of the point. The direction of the line can be expressed by giving the change in the x and y coordinates of this line along a given length of it. Earlier, we calculated this change by subtracting the starting x,y coordinates from the ending x,y coordinates. In a polar coordinate system, however, the starting coordinates will always be 0,0, since the line extended to a point in the system always starts at the origin. So the direction vector for the line can simply be expressed as the coordinates of the point to which it extends: 7,5.

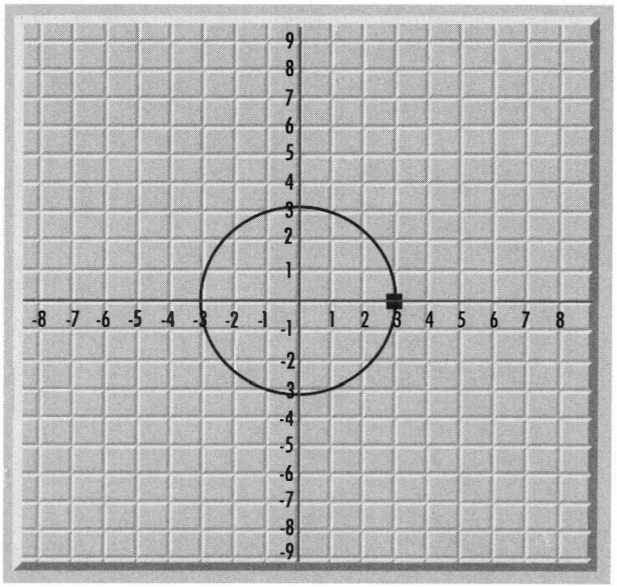


Figure 7-22 A polar coordinate system superimposed over a Cartesian coordinate system

We can calculate the magnitude of the vector using the Pythagorean method that we used earlier. Once again, however, it's not necessary to first subtract the starting from the ending coordinates of the line to get its length, because it starts at 0,0. So we can just use the x,y coordinates of the point as the x,y lengths of the line. We can measure the distance from the origin to the point at 7,5 with the calculation

```
distance = sqrt(7*7+5*5)
```

or

```
distance = sqrt(49+25)
```

or

```
distance = sqrt(74)
```

or

```
distance = 8.602
```

Thus the magnitude of the vector between the origin and the point at 7,5 is 8.602. And the direction of the vector, expressed in terms of the Cartesian coordinate system, is 7,5. Thus we can describe the point in polar terms as being at a distance of 8.602 from the origin along a vector of 7,5. This may seem like

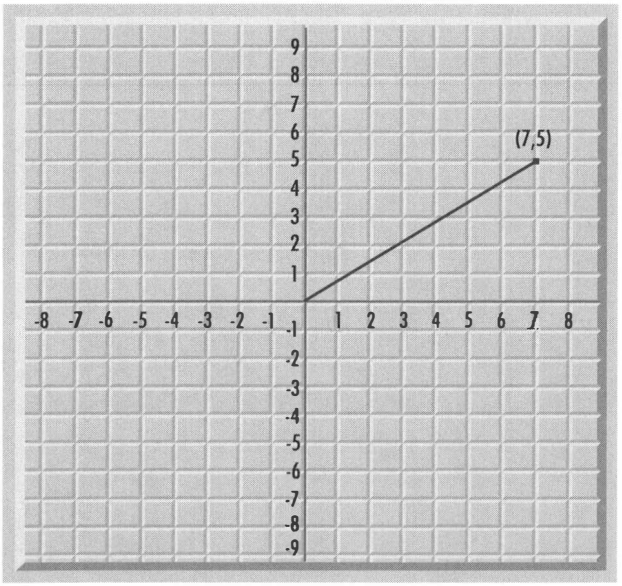


Figure 7-23 A point at coordinates 7,5 from the origin

an unnecessary amount of information, since merely the numbers 7,5 suffice to express the position of this point within a Cartesian coordinate system. But bear in mind that we can also express the vector as any equivalent ratio: 14,10 or 28,20 or even 3.5,2.5. And this flexibility allows us to perform translations from vectors to angles. To see how, let's look at the concept of the unit vector.

The Unit Vector

A line can have any positive magnitude and any possible direction vector. There is, however, one special direction vector associated with a special length of line (or portion of a line). It is the vector measured along a single Cartesian unit of the length of a line and it is known, logically enough, as the unit vector. It has special properties, a few of which we are going to mention here.

Any vector can be converted into a unit vector simply by dividing both the x and y components of the vector by the magnitude of the line along which the vector was originally measured. This is called *normalizing* the vector. It is done like this:

```
normalized_x = x / magnitude;
normalized_y = y / magnitude;
```

where x and y are the x and y coordinates of the end of a line that starts at the origin, magnitude is the distance from the origin to the end in Cartesian units,

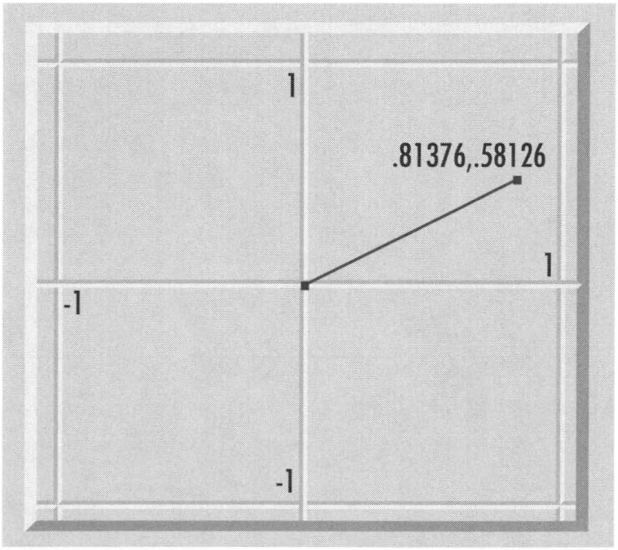


Figure 7-24 A unit vector of .81376,.58126

and normalized_x and normalized_y are the normalized coordinates of the line's direction vector.

For instance, the vector to the point in Figure 7-23 was 7,5 and the magnitude of the vector was 8.602, so we can normalize this vector by dividing both the x and y components of the vector by 8.602, like this:

```
normalized_x = 7 / 8.602;
normalized_y = 5 / 8.602;
```

If you performed these calculations, you would learn that the unit vector for the line that goes from the origin of the Cartesian coordinate system (or, for that matter, *any* Cartesian coordinate system) to the point at coordinates 7,5 is .81376,.58126. (See Figure 7-24.)

The Unit Circle

If we draw all the possible points that are one unit away from the origin of a Cartesian coordinate system, as in Figure 7-25, they will form a neat circle around the origin. The circle is called a unit circle, because its radius is precisely one Cartesian unit. Every possible line extending one or more units from the origin crosses the unit circle. And if we measure that vector of a line along the portion of the line that extends from the origin to the unit circle, we will obtain — obviously — a unit vector. If you think about it for a second, you'll see that

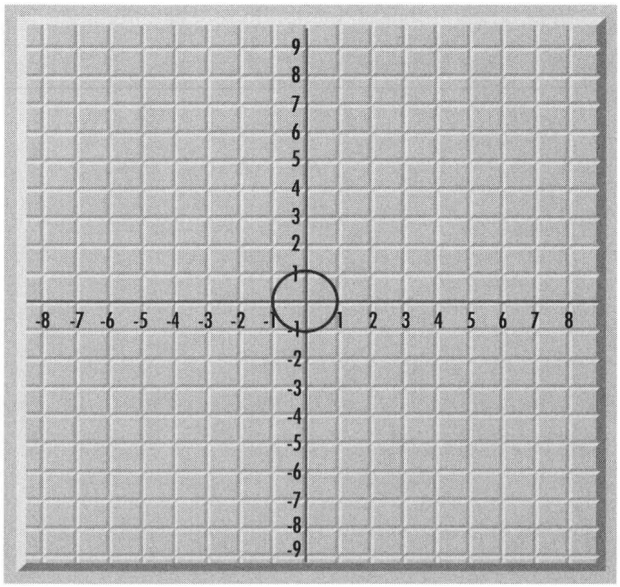


Figure 7-25 A unit circle around the origin of a Cartesian coordinate system

the x,y components of this unit vector are also the Cartesian coordinates for the point at which the line crosses the unit circle.

Rotating a Point

For those of you who didn't fall completely asleep during the foregoing burst of heavy mathematics, I have a (possibly) unexpected gift. I am going to show you how to use the coordinates at which a line crosses the unit circle to create your own three-dimensional world. This may seem like a pretty spectacular bonus to receive from a pair of numbers in a Cartesian coordinate system, but the concept of the normalized coordinates of a unit vector are at the heart of every flight simulator and virtual reality program on the market. The normalized coordinates of a vector — the coordinates at which the vector crosses the unit circle — can be used to translate back and forth between angles measured as degrees (or radians) and Cartesian direction vectors. So special are the x and y components of the normalized direction vector that they have been given names: They are the cosine and sine, respectively, of the angle at which the vector extends from the origin.

Even if you slept through trigonometry (if not through the first part of this chapter), you probably remember the terms cosine and sine. However, you probably remember them in terms of right angles. After all, the very term

"trigonometry" comes from an outmoded word meaning "triangle." And, indeed, the sine and cosine we have stumbled upon at the point where a vector crosses the unit circle are the selfsame sine and cosine that you learned about in trig class. In fact, if you remember enough of your trig, you can probably see how these two different ways of looking at sine and cosine are actually the same, since it's pretty easy to find right triangles in a Cartesian coordinate system (as we did a moment ago, when we used the Pythagorean theorem to calculate the magnitude of a vector). However, we're not going to discuss the theory of sines and cosines at any greater length in this chapter — you've had enough mathematics to stupefy a rhinoceros.

Standard mathematical algorithms for calculating sines and cosines can be found in any book on trigonometry. We won't be going into them here because our C/C++ compiler is supplied with standard library functions (the prototypes for which are in the header file MATH.H) for calculating them. These library functions are slow, but they're good enough for the demonstration programs that we'll be writing in this book. Later we'll develop our own high-speed sine and cosine routines (though we won't use the standard mathematical algorithms for doing so).

For our purposes, the most important thing about sines and cosines is that they allow us to translate from an angle to Cartesian coordinates in space. And this in turn will allow us to take a point at a given set of coordinates relative to the origin and rotate it around the origin by any number of degrees or radians. (As it happens, the standard library sine and cosine functions that come with our C/C++ compiler expect an angle to be measured in radians.)

And this leads us to the topic with which we will finally wrap up this chapter on mathematics: coordinate transformations.

Coordinate Transformations

We will generally deal with three- (and two-and-a-half-) dimensional scenes in terms of the Cartesian coordinates of points within those scenes. As we saw earlier in this chapter, a polygon-fill flight simulator deals with three-dimensional scenes in terms of the three-dimensional x,y,z coordinates of the vertices of polygons. As we'll see in the *next* chapter, a maze can be dealt with in terms of the two-dimensional x,y coordinates at which walls and objects are placed within the maze.

In both flight simulators and maze games, the secret of creating the illusion of movement is to move a point realistically in two and three dimensions. Moving a point within two- and three-dimensional coordinate systems is called *transforming* the point. There are primarily two ways in which we will want to transform points: the first is to translate them and the second is to rotate them.

Translating a Point

Translating a point means, simply, to move it from one place in the coordinate system to another along a straight line. Generally, a transformation is expressed in terms of how many Cartesian units we want the x and y coordinates of the point to change during the translation. Actually translating a point is a simple matter of adding these values to the current x,y coordinates of the point, like this:

```
new_x = old_x + x_trans;
new_y = old_y + y_trans;
```

where *old_x,old_y* are the original coordinates of the point, *trans_x,trans_y* is the distance that we wish to move the point from its original position, and *new_x,new_y* are the coordinates of the point after the translation. For instance, in Figure 7-26, the point 7,8 has been translated by -2,3 units to 5,11.

Rotating a Point

Rotating a point is somewhat more complex than translating it. However, there are standard formulas for doing so, based on the sine and cosine functions. Here, for instance, is the formula for rotating a point to a given angle relative to the origin of a Cartesian coordinate system:

```
new_x = old_x * cosine(ANGLE) - old_y * sine(ANGLE);
new_y = old_x * sine(ANGLE)+ old_y * cos(ANGLE);
```

where *old_x,old_y* are the original coordinates of the point, ANGLE is the angle that we wish to rotate it around the origin (expressed in whatever unit the sine

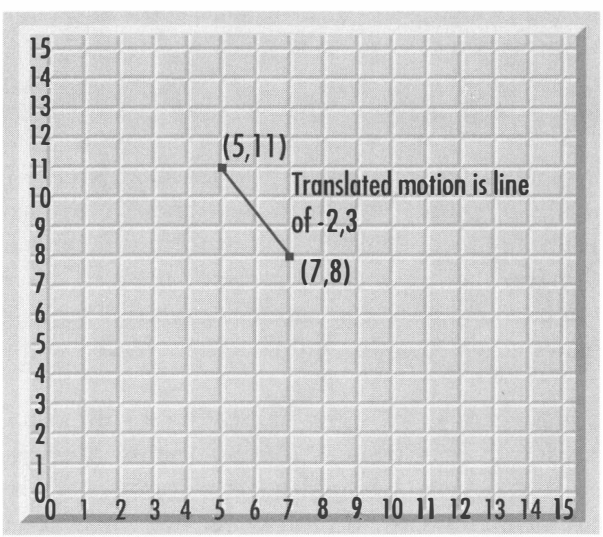


Figure 7-26 Translating a point

and cosine functions are expecting), and *new_x,new_y* are the coordinates of the point after the rotation. This rotation is always performed around the origin of the system. If you wish to rotate a point around some point other than the origin of the system, you must first translate that point relative to the origin by the amount that the center of the rotation is displaced from the origin. That's a simple matter of subtracting the coordinates of the center of rotation from the coordinates of the point (or adding the negative of the coordinates, which comes to the same thing). Then perform the rotation as described above and add the displacement back to the newly rotated coordinates. Here is one way in which this might be done:

```
temp_x = old_x - center_y;
temp_y = old_y - center_y;
new_x = temp x * cosine(ANGLE) - old_y * sine(ANGLE);
new_y = temp_x * sine(ANGLE)+ - old_y * cos(ANGLE);
new_x += center_x;
new_y += center_y;
```

where *center_x* and *center_y* are the coordinates of the point around which we wish to rotate the point at *old_x,old_y* by ANGLE, and *temp_x,temp_y* are temporary value holders. The rotated position of the point will once again be at *new_y,new_x*.

Three-Dimensional Rotation

We can also rotate a point in a three-dimensional coordinate system, but this is really just a matter of performing three separate two-dimensional rotations. There are three different axes in a three-dimensional coordinate system — x, y, and z — and any two of these can be said to describe a two-dimensional plane. When we perform a two-dimensional rotation, as we did above, we are rotating the point in the x,y plane, that is, the plane formed by the x and y axes. But we can also rotate points in the x,z plane (the plane formed by the x and z axes) and the y,z plane (the plane formed by the y and z axes). To rotate an object three-dimensionally, we must rotate it separately in each of these planes. (We can also think of this as rotating the point around each of the three coordinate axes, where rotation in the x,z plane is rotation around the y axis; rotation around the x,y plane is rotation around the z axis; and rotation around the y,z plane is rotation around the x axis.)

The amount that this point is to be rotated should be expressed by three angles: the angle for rotation around the x axis (rotation in the y,z plane), the angle for rotation around the y axis (rotation in the x,z plane), and the angle for rotation around the z axis (rotation in the x,y plane). Here is a formula for rotation around the x axis:

```
new_y = old y * cosine(ANGLE) - old_z * sine(ANGLE);
new_z = old_y * sine(ANGLE) + old_z * cosine(ANGLE);
new_x = old_x;
```

Here is the formula for rotation around the y axis:

```
new_z = old_z * cosine(ANGLE) - old_x * sine(ANGLE);
new_x = old_z * sine(ANGLE) + old_x * cosine(ANGLE);
new_y = old_y
```

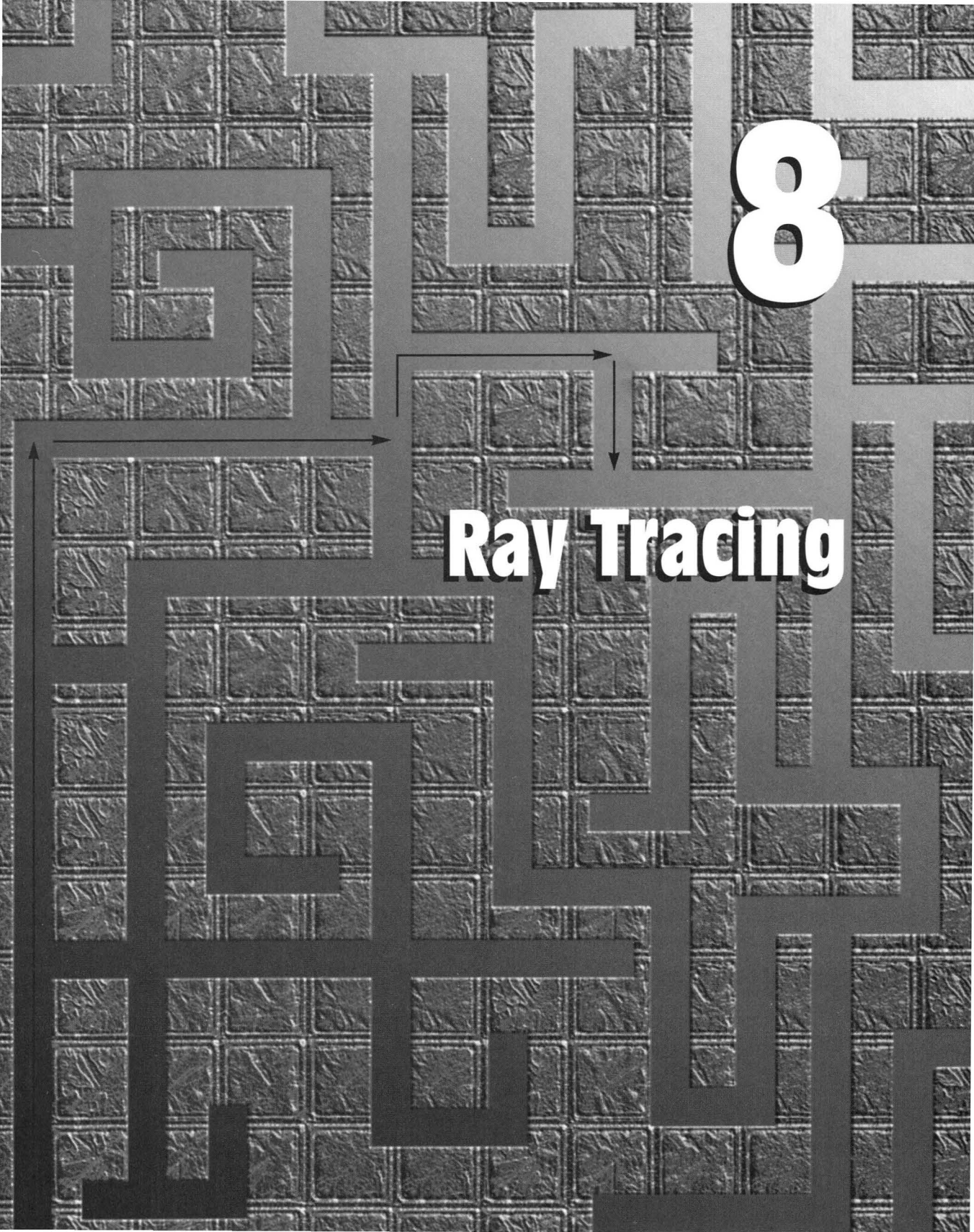
And here is the formula for rotation around the z axis:

```
new_x = old_x * cosine(ANGLE) - old_y * sine(ANGLE);
new_y = old_x * sine(ANGLE) + old_y * cosine(ANGLE);
new_z = old_z;
```

You'll notice that the formula for rotation around the z axis is identical to our earlier formula for two-dimensional rotation, since this is also rotation in the x,y plane that we normally use for two-dimensional Cartesian coordinates.

Let 'Er Rip

That's enough mathematical background for now. We'll pick up additional math as we go along, building on the concepts that were introduced in this chapter. Now that we have the ability to translate points in two- and three-dimensional coordinate systems, we can start writing some two-and-a-half-dimensional maze-generation code, which will simulate three-dimensional motion through a maze without quite going all the way to the third dimension. In the next chapter, we'll stumble our way into the newest, most exciting frontier of all: ray-cast graphics.


8
Ray Tracing

8

In the more than two decades since Nolan Bushnell unleashed Pong on an unsuspecting world, video games have become increasingly more realistic. Pong featured a large white pixel being batted back and forth by a pair of horizontal lines, in a crude simulation of Ping-Pong. If the game were rereleased today, it would probably feature a pair of animated players knocking a high-resolution ball back and forth across a texture-mapped table. Two decades from now, a hologram of the two players may pop out of the computer and into your living room. Watch out for that backstroke!

Maze games have not escaped this tidal wave of increased realism. Players demand maze graphics that are both vivid and fast. The challenge facing us as programmers is to come up with a method for depicting exceptionally vivid mazes that is not so complicated that it slows the resulting animation to a crawl. Unfortunately, realism and speed are often incompatible. By and large, the more realistic computer graphics are, the longer it takes to draw them.

So for the moment, let's forget about speed. Let's find the most realistic possible method of generating computer graphics, then find a way to speed it up. We may sacrifice a few elements of realism when we start stripping our maze-generation algorithms down for speed. But, if we play our cards right, the game players will never notice.

Soaking Up Rays

What's the most realistic method of creating computer graphic images? That's hard to say. There are several techniques now available for generating realistic images on a computer screen, and you can easily start an argument among graphics programmers by asking which is best. However, the best-known method of generating realistic computer graphics is almost certainly ray tracing.

The very name "ray tracing" has taken on nearly magical connotations in recent years. Even people who don't know a floppy disk from a frisbee can spot the ray-traced animation sequences in movies like *Beauty and the Beast*. Ray-traced images seem to glow with a vividness that is almost otherworldly.

Surely a program that creates such wonderful graphics must itself be an amazingly complex piece of code. Only the most brilliant programmers could possibly dream of writing such a program, right?

Well, no. Ray tracing may create beautiful images, but the ray tracers themselves are marvelously simple, at least in concept. Some complications result when programmers add clever tricks to the ray-tracing code in order to make the images render more quickly. But the basic algorithm underlying ray tracing is very straightforward and easy to understand. To prove it, we'll take a look at how ray tracers work.

Looking at the World

Take a look at the world around you. What do you see? Tables and chairs? Trees and houses? A desk with a microcomputer on top?

In a sense, the answer is: None of the above. No matter what you think you are seeing, what you are *really* seeing is rays of light.

Sight is the most intense of human senses, the one to which the largest portion of our brain is devoted. Sight begins when rays of light pass through the lenses of our eyes and strike photosensitive nerves inside them. These nerves pass information about those rays to our brain, which process that information to produce the images that we see. All of the information in that image comes from rays of light that are either bouncing off or being created by the objects at which we are looking.

This may seem like an obvious statement, yet it's easy to forget the important role that light plays in our visual system. Objects such as the sun and incandescent lamps produce rays of light. These rays are colored by certain properties of the objects producing them. The rays then proceed to strike the surfaces of other objects, at which point they are either absorbed by the surfaces (causing those surfaces to become warmer) or reflected by them. Actually, most surfaces both reflect *and* absorb light. Because this absorption is usually selective — only certain colors of light are absorbed — the color of the reflected light is

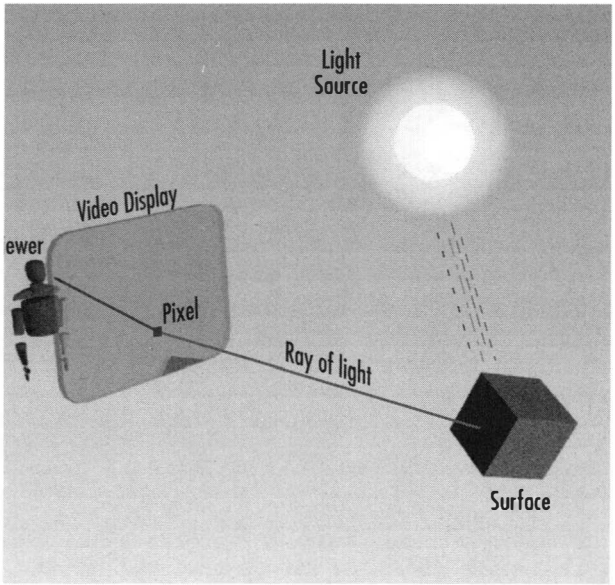


Figure 8-1 A ray being traced backward from the viewer into the world inside the computer

usually different from the color of the light that originally struck the surface. When those reflected rays of light reach our eyes, we see only the colors that were *not* absorbed, which is why most surfaces in our field of view have colors other than those of the object producing the light.

For instance, when white light from an incandescent bulb strikes the surface of the filing cabinet sitting next to my desk, the surface absorbs the colors green and blue, leaving only red. Which is why the cabinet appears to be red when I look at it. If I were to place a colored filter over the nearest lamp to filter the red out of the light, the cabinet might well appear nearly black, because it would absorb all of the colors that managed to get through the filter.

In order to produce realistic images, a ray tracer tracks rays of light through a computer-simulated "world" and determines what color those rays would have after reflecting off of various surfaces. It then uses this information to determine the color of the pixels on the display. However, ray tracers don't attempt to trace every ray of light leaving every light source in the simulated world, or even a representative sample thereof. That would take much too long. Instead, ray tracers work backward. They start at the eyes of the viewer sitting in front of the computer and trace rays of light backward into the computer display (see Figure 8-1).

How many rays of light does a ray tracer actually trace? Usually, a single ray for every pixel in the image being produced. One by one, the ray tracer traces

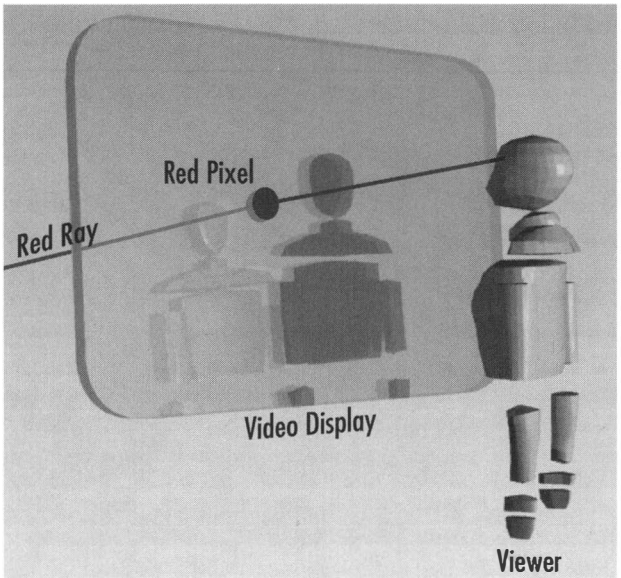


Figure 8-2 Each pixel on the display is given the color of the light ray that passes through it

rays backward from the viewer's eyes through each pixel on the display into the simulated world inside. The pixels on the display are then given colors based on the color of the light source that initially produced the ray minus any colors absorbed by surfaces or filters that it encountered on the way (see Figure 8-2).

Ray tracing is just about the simplest method ever developed for creating a graphic image, hyper-realistic or otherwise. Because each ray of light is tracked separately, it is not necessary to worry about inadvertently displaying hidden surfaces in the image or clipping portions of the image against the edges of the viewport. In fact, ray tracing has only one major drawback: It is slow. Tracing all of those rays takes time. There's no way that any ray tracer in existence today could be used to produce animated computer images on the fly. Nonetheless, let's take a closer look at ray tracing, then figure out how we can use some of the basic principles of the technique in a high-speed animation engine. In order to get an inside look at ray tracing, we'll spend the rest of this chapter developing an actual working ray tracer.

A Ray Tracer of Our Own

The first issue that we need to deal with in building a ray tracer is the way in which we will store the simulated world that we want to trace. For the simple ray

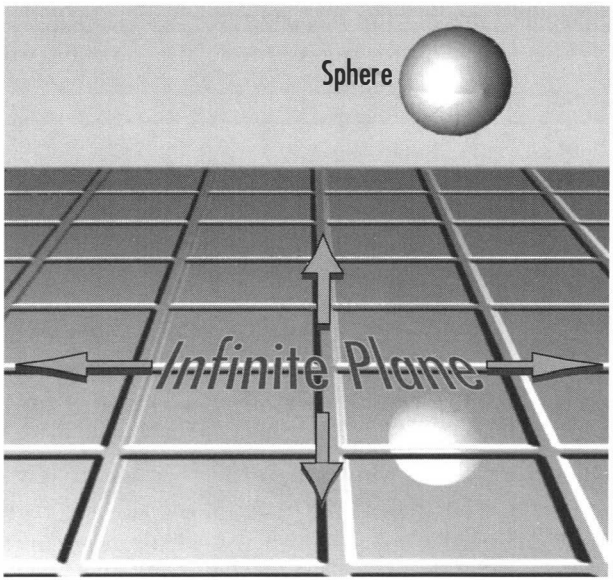


Figure 8-3 The two types of objects supported by our ray tracer

tracer that we're about to write, there's no need to get fancy. We'll store our "world" as a simple one-dimensional array, in which each element contains a description of one object in the world. The objects themselves will be simple shape primitives that can be described by a few parameters. To keep things as simple as possible, our ray tracer will only support two types of shapes: spheres and infinite planes (see Figure 8-3).

Even this simple arrangement, however, introduces some complications in our program. The information needed to describe an infinite plane is quite different from the information used to describe a sphere. Thus we'll need to use the type of C/C++ structure known as a *union* to cram two different types of structures, or pointers to those structures, into a single element of the array. So we'll begin by creating separate structures for each type of object, then create a single type of pointer that can point at both of them.

Before defining either of these structures, we'll define a special type of structure to hold numerical triplets, which we'll be using in a couple of ways in our object definitions. We'll refer to this type of structure as *vector_type:*

```
struct vector_type {
  double x,y,z;
};
```

One obvious use for this structure will be storing three-dimensional spatial coordinates, as described in the last chapter. Another use will be for describing

the spatial orientation of certain surfaces, as you'll see in our next definition.

The definition for the infinite plane looks like this:

```
struct plane_type {
  vector_type surface_normal;
  double distance;
};
```

The *vector_type* structure here represents the so-called *surface normal* of the plane. The surface normal is a line perpendicular to (or "normal to," as a mathematician might put it) the surface of the plane. It can be represented by a three-number vector representing the change in coordinate values along a unit of the line, as shown in Figure 8-4. (See chapter 7 for a detailed discussion of vectors.) This vector gives us the orientation of the plane in space. Since the plane is infinite — that is, it stretches away forever in two dimensions — the only other thing we need to know about it is where it is along the length of the surface normal. If we imagine the surface normal as a line stretching forever through space along one dimension, then an infinite number of infinite planes could be dispersed along its length. Thus we further define the plane with the variable *distance*. If we imagine that the surface normal intersects the 0,0 origin

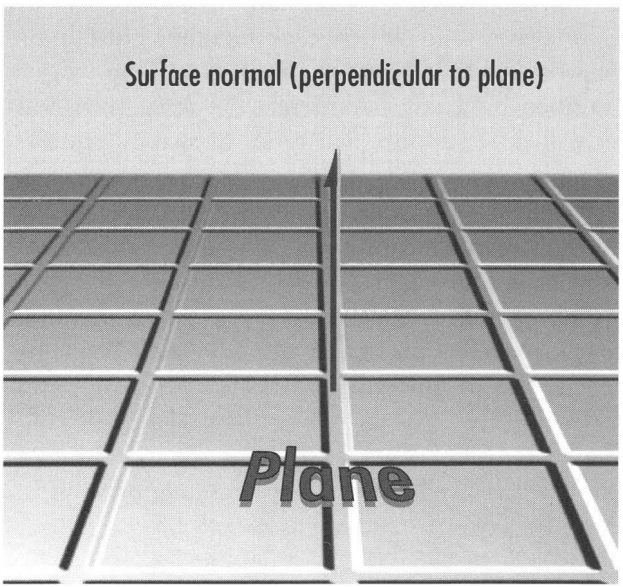


Figure 8-4 The surface normal of a plane

point of our universe at some point in its infinite length, then *distance* represents the number of units from the the origin of the universe to the surface of the plane (see Figure 8-4). With those two values — the surface normal and the distance from the origin — we have completely described the infinite plane.

We don't need to know the orientation of the sphere in space because a sphere, being symmetrical in all dimensions, has no real orientation. Put another way, an upside-down sphere looks exactly like a rightside-up one. (This is not true if the sphere has some sort of pattern mapped onto its surface, but we won't be using any surface patterns in our ray tracer.) What we need to know is the spatial location of the sphere — that is, the coordinates of the sphere's center. It's traditional in writing a ray tracer to use variables called *l, m,* and *n* to represent these coordinates. We also need to know the radius of the sphere, in coordinate units. Here's the definition for the sphere structure:

```
struct sphere_type {
  double l,m,n;      // Center of sphere
  int r;              // Radius of sphere
};
```

Now that we have these two structures, we'll use them to create a type of variable that can point to both of these structures:

```
union object_union {
  plane_type *plane;
  sphere_type *sphere;
};
```

Finally, we'll create a structure that can be used to define an object in the ray tracer's world:

```
struct object_type {
  int type_of_object;
  int color;
  object_union obj;
};
```

The first field, *type_of_object,* is a number defining what type of object the structure represents. We'll need to define constants representing the two types of objects that we will be tracing:

```
const SPHERE=1;
const INFINITE_PLANE=2;
```

If the *type_of_object* field is set to 1, the object is a sphere. If the *type_of_object* field is set to 2, the object is an infinite plane. Other values are not defined, though this ray tracer could easily be expanded to recognize other object types.

The *color* field will define the color that should be given to the surfaces of the object when they are fully illuminated. In practice, this will represent the maximum possible color of the surface; in most instances, the surface will be shown at least partially in shadow and won't achieve this maximum color. We'll have more to say about color in a moment.

Finally, the *obj* field can be either a pointer to a *plane_type* structure or a *sphere_type* structure, depending on which we want it to represent. Thus we define it as an *object_union* structure.

Low-Resolution Rays

Before we define the actual objects in the ray-traced world, let's talk a little about how we're going to set up the video display. Since the game that we're going to write in this book will use the 320 by 200 by 256-color VGA graphics mode, the simple ray tracer that we'll develop in this chapter will use that mode too. Unfortunately, a ray-traced image usually requires a lot more than 256 colors, so we'll simplify our task by limiting the number of colors in the images that we produce. Instead of a full-color ray trace, we'll create a gray-scale ray trace. The VGA adapter can generate 64 different shades of gray, enough to create some nicely detailed and subtly shaded images.

We'll need to create a custom palette to hold these shades of gray. We'll store that palette in a *char* array, defined like this:

```
char pal[768];
```

Then we'll put some code in the initialization portion of the ray-tracer program that will generate a gray-scale palette. In an RGB color system like the one used by the VGA adapter, all colors in which the levels of red, green, and blue are equal will appear to be gray. Since the VGA adapter can generate 64 possible shades of each color component, it can display 64 possible shades of gray, from 0,0,0 to 63,63,63. Here's a *for()* loop that will put 64 shades of gray into the first 64 elements of a palette array:

```
for (int grey=0; grey<64; grey++) {
  pal[grey*3] = (int)grey;
  pal[grey*3+1] = (int)grey;
  pal[grey*3+2] = (int)grey;
}
```

This loop iterates 64 times, one for each shade of gray. Since each palette entry is 3 bytes long, the actual index of each palette entry is found by multipying the loop index by 3. Only the first 64 palette colors are affected by this loop. We'll simply ignore the rest.

Defining the World

Now that we have the necessary structures and a gray-scale palette, we can define the actual spheres and planes that will appear in the ray-traced image. Normally, this information would be placed in a file separate from the ray tracer itself, to facilitate the design of new images. But we don't have space in this chapter to construct the sort of parser logic that would be necessary for our ray tracer to read data in a separate file. Instead, we'll define the image right in the ray-tracer program itself. To trace a different image, you'll need to alter the file RTDEMO.CPP and recompile the ray tracer using the RTDEMO.PRJ project file.

We'll start by definining some constants. The first will tell us how many objects are in the world:

```
const NUMBER_OF_OBJECTS=2;
```

To keep things simple, the first world that we trace will have only two objects in it. Next we'll define the background color:

```
const BACKGROUND_COLOR=16;
```

The background color is the color given to those rays of light (and the pixels that they represent) that do not intersect either an object or a light source within the world. This color will appear in the background of the image, behind any objects. We have defined it here as VGA color 16, which in our gray-scale palette will be a fairly dark shade of gray.

Next, we'll define the intensity of the ambient lighting in the scene. Ambient lighting is a diffuse, usually white, light that seems to come from all directions at once (see Figure 8-5). In the real world, it is light that has been scattered randomly after being reflected off many different objects, including molecules of air. Ambient lighting is the reason that shadows in the real world are rarely completely black (though in airless environments, such as the surface of the moon, ambient lighting is a much less important factor than it is here on earth and shadows are very nearly lightless). If it weren't for ambient light, the unlit portions of objects would be pitch black, something that usually only happens in completely darkened rooms. Ray tracers don't have the time to track all of these diffuse rays of light, so we need to define an intensity level for the ambient lighting that will automatically be used on all objects in the scene. We'll represent the ambient level as a fraction between 0 and 1, which we'll arbitrarily set to .27:

```
const double AMBIENT_INTENSITY=.27;
```

Of course, there must be at least one real light source in the scene. If every surface was given only the ambient color, the image that we produced would be quite boring. But in the interests of keeping this ray tracer as simple as possible,

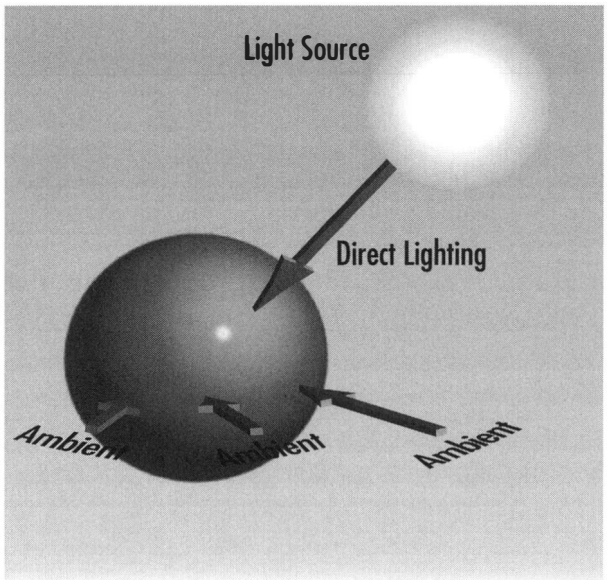


Figure 8-5 Ambient lighting is reflected light that seems to come from all directions at once

we'll allow one and only one light source in the scene. And we'll need to create a *vector_type* constant in which we'll store the x,y,z coordinates of the light source within our imaginary world:

```
const vector_type LIGHT_SOURCE={500,-200,150};
```

Now let's define the objects in the world. For our first trace, we'll only have two objects in the world, a sphere and an infinite plane. Here's how we'll initialize the plane:

```
struct plane_type plane={ {0,-1,0},       // Surface normal
                  500                      // Distance from origin
};
```

This tells us (and the ray tracer) that the surface normal is pointed in a perfectly vertical direction, parallel with the y axis of the universe, as in Figure 8-6. (That's because the vector — 0,-1,0 — used to define the plane only changes in the y direction.) It also tells us that the plane is 500 coordinate units away from the universal origin along this vector.

Here's how we define the sphere:

```
struct sphere_type sphere={0,40,300,     // Coordinates of center
                           100};         // Radius
```

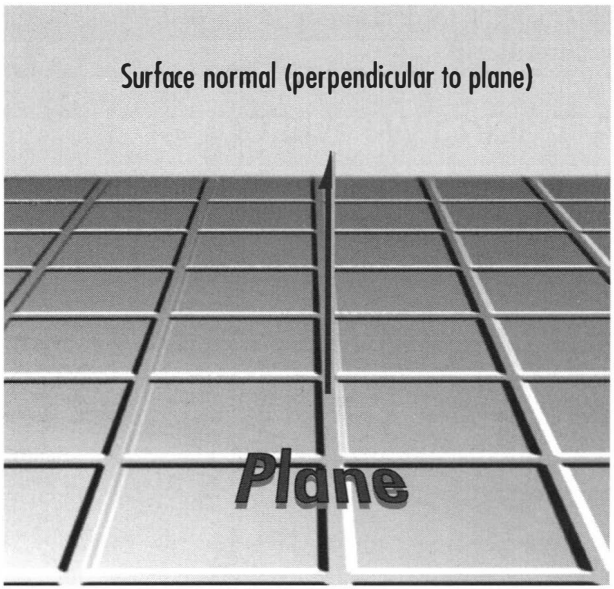


Figure 8-6 The normal of the plane is pointed in a perfectly vertical direction

This tells us that the sphere has a radius of 100 units and that the center of the sphere is located at universal coordinates 0,40,300.

Next, we need to create an array of *object_type* structures to point to these objects. Here's the declaration of that array:

```
struct object_type objects[NUMBER_OF_OBJECTS]={
  {SPHERE,56,(plane_type *) &sphere},
  {INFINITE_PLANE,48,&plane}
};
```

The number of elements in the array is two, since that's how we've defined the NUMBER_OF_OBJECTS constant. The first element is set to point at the structure defining the sphere. Despite the use of the *union* statement in defining the pointer to the object, it's necessary to use a type override (in parentheses) to point it at the sphere structure. The type override corresponds to the first type used in the union; without the override, the compiler would complain that it couldn't perform the necessary data conversion. This declaration also tells us that the color of the sphere will be 56, which corresponds to the 56th shade of gray in our gray-scale palette, the one that we define as having RGB values 56,56,56.

The second element points at the infinite plane structure. No override is necessary here, since this was the first type of structure defined in the *union*

statement. This declaration also tells us that the plane is to be colored with gray-scale shade 48.

A-Tracing We Will Go

Now let's start tracing rays. Because we're going to need some random numbers later on — you'll see why in a moment — we'll begin the *main()* function of the ray tracer by initializing the random number generator:

```
randomize();
```

Then we'll initialize the gray-scale palette, using the *for()* loop that we developed earlier in this chapter. Next, we'll run through our standard mode 13h setup, setting the screen mode, saving the old screen mode, and so on. And we'll set the palette to the gray-scale palette that we've developed:

```
setpalette(pal,0,256);
```

We'll be needing a couple of *vector_type* variables, so we'll declare them here:

```
vector_type rayvec,raystart;
```

The actual ray tracing will be performed inside a pair of nested *for()* loops, one to work through all of the rows on the screen and the other to work through all of the pixels in a row. The index variables for these loops will be called *yscreen* and *xscreen,* respectively:

```
for (int yscreen=0; yscreen<200; yscreen++) {
  for (int xscreen=0; xscreen<320; xscreen++) {
```

Follow That Ray

Now we need a ray to trace. All that the ray tracer needs to know initially about a ray is where it starts and the direction it's heading. Both of these things can be expressed by vectors. We'll use the *vector_type* variables *raystart* and *rayvec* to hold this information. The variable *raystart* will contain the x,y,z coordinates of the start of the ray, and the variable *rayvec* will contain the change in the x,y,z coordinates along a given length of the ray. (For a detailed discussion on vectors, see chapter 7.)

It's important here to remember that a ray tracer tracks rays *backward.* Thus, while the rays that we track actually begin at a light source, we have to trace them as though they started at the eyes of the viewer. And where are the viewer's eyes? Although a full-service ray tracer would allow us to place the viewer at any point in the world, looking in any direction, our simplified ray tracer will only allow the viewer to be placed at the origin of the world coordinate system, at

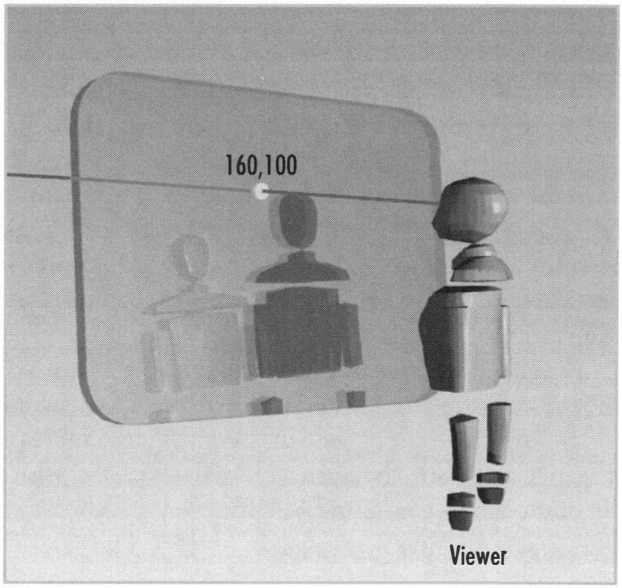


Figure 8-7 The viewer's eyes are aligned with pixel position 160,100 at a distance of 100 units from the screen

coordinates 0,0,0, looking straight down the z axis into the display. We'll further assume that the viewer's actual physical position places him or her a distance of 100 coordinate units from the video display, looking directly at the pixel in the center of the display, as in Figure 8-7. (Actually, because the number of pixels in both the x and y directions on the display is even, there is no pixel that is precisely in the center of the display. So we'll arbitrarily designate the pixel at coordinates 160,100 as the center pixel.) Let's store these figures in some constants, because we'll be needing them later:

```
const VIEWER_X=160;
const VIEWER_Y=100;
const VIEWER_DISTANCE=100;
```

It would be fair to assign a value of 0,0,0 to the *vector_type* variable *raystart,* since the ray "starts" at the viewer's eyes. But those numbers could cause problems later on, such as the dreaded divide-by-zero error. It's just as fair to use the point where the ray intersects the computer's screen as the starting point. The coordinates of this point are in the loop index variables *screenx* and *screeny.* However, we'll first subtract VIEWER_X and VIEWER_Y from *xscreen* and *yscreen,* to make these coordinates relative to the viewer's position:

```
raystart.x=xscreen-VIEWER_X;
raystart.y=(yscreen-VIEWER_Y)*1.2;
```

Why are we multiplying the y position by 1.2? In the 320 by 200 VGA mode, pixels on the display are not perfectly square. They are stretched by a small amount in the horizontal direction, making them 20 percent wider than they are tall (see Figure 8-8). This has the effect of making ray traced or otherwise computer-generated images appear a little square on the display. Multiplying the vertical coordinate by 1.2 compensates for this stretching, effectively unsquashing the images.

We also need to determine the pixel's z coordinate relative to the viewer. Since the viewer is looking straight down the z axis in the positive direction — that is, in the direction along which z coordinates increase — this will be equal to the distance between the screen and the viewer, which we've already stored in the constant VIEWER_DISTANCE:

```
raystart.z=VIEWER_DISTANCE;
```

We could also use these three values for *rayvec,* the direction vector for the ray, because they represent the distance that the ray has traveled along the segment of the ray from the viewer's eyes to the screen. However, many of the equations used in ray tracing require that the vector of the ray be expressed as a unit vector. You'll recall from chapter 7 that a unit vector is the change in coordinates along a

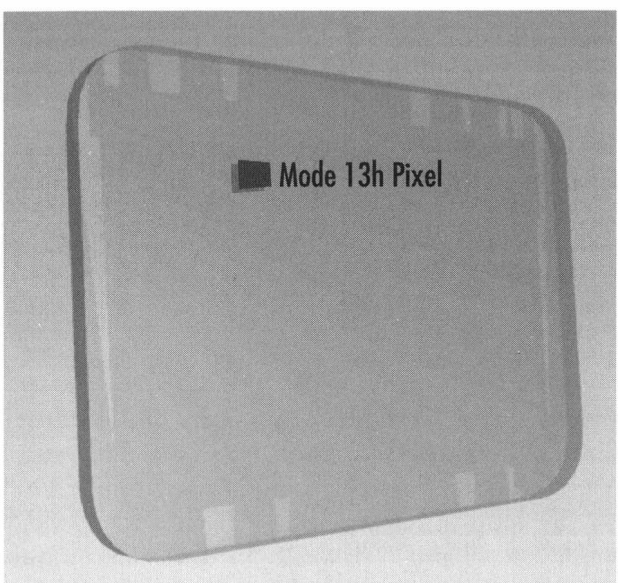


Figure 8-8 Pixels on the mode 13h screen are wider than they are tall

segment of a line equal in length to one unit of the coordinate system. It can be obtained by taking the vector of the line along *any* length, then dividing the x, y, and z components of the vector by that length.

We can calculate the length of the ray from the viewer's position at 0,0,0 to the pixel at *xscreen,yscreen* by using a three-dimensional version of the Pythagorean distance equation introduced in chapter 7:

$$(D = \sqrt{(\sqrt{a^2 + b^2})^2 + c^2})$$

We can imagine that the distance between the viewer and the screen is one side of a right triangle, while the distance from the center of the screen to the pixel is another side of that triangle. The distance from the viewer to the pixel, then, becomes the hypotenuse of that triangle (see Figure 8-9). We can calculate the distance like this:

```
double distance=sqrt(raystart.x*raystart.x+raystart.y*raystart.y
                     +raystart.z*raystart.z);
```

Then we can set *rayvec* equal to the unit vector between the viewer and the display like this:

```
rayvec.x = raystart.x/distance;
rayvec.y = raystart.y/distance;
rayvec.z = raystart.z/distance;
```

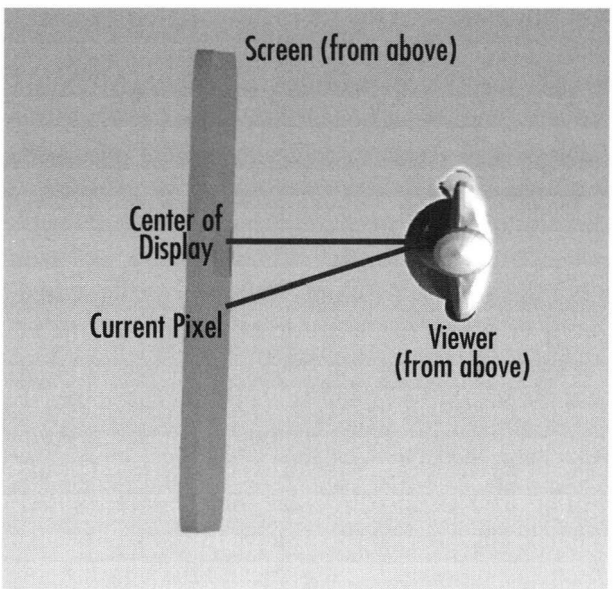


Figure 8-9 The screen, a line from the viewer to the center of the display, and the ray from the viewer's eye form a right triangle

The ray tracing will be performed by a function called *trace_ray(),* which will take a pair of vectors as arguments and will return a gray-scale color value. This value will in turn be passed to a second function called *plot(),* which will take a pair of x,y screen coordinates and a gray-scale value as arguments. Since the *trace_ray()* function returns a gray-scale value, we can pass the result of that function directly to *plot(),* like this:

```
plot(xscreen,yscreen,trace_ray(rayvec,raystart));
```

All that remains for us to do in the *main()* function is to close the two *for()* loops, pause until the user hits a key, and reset the old video mode.

The Heart of the Ray Tracer

Obviously, the real heart of the ray tracer is in the *trace_ray* function. This function will loop through all of the objects in the world — at the moment, there are only two — and will perform calculations to see if the ray intersects the surface of any of those objects. If the ray intersects an object, the angle between the surface intersected by the ray and the nearest light source will be calculated. This angle will be used to calculate a fractional value between 0 and 1 that can be multiplied by the gray-scale value of the surface to obtain the color of the ray. (The AMBIENT_INTENSITY value will be added to this fraction before the multiplication takes place.) If the angle is greater than 180 degrees, the surface is facing away from the light source and *only* the AMBIENT_INTENSITY is used to calculate the color of the ray. (If AMBIENT_INTENSITY is 0 — that is, if there is no ambient light in the scene at all — the ray will be assigned a gray-scale value of 0, meaning that it is completely in shadow and should appear black in the scene.) The color of the ray is passed to the calling function, where we plot the current screen pixel with that color.

As simple as all of this is in concept, it will require some fancy mathematics to pull off. Fortunately, there are books available that provide most of the necessary math, so there's no need for us to reinvent the wheel. Several of these books are listed in the back of this book, if you want to extend the simple ray tracer in this chapter into a full-fledged rendering program. Be warned, however, that these books can be pretty heavy going for the nonmathematically minded. (They've already given the author of this book several headaches.)

The first thing the *trace_ray()* function will do is look for intersections between the ray and the objects in the scene. This requires that we find a pair of equations representing the ray and the object being tested.

Since all of the objects in the scene are simple geometic primitives — spheres and planes — we can use standard math equations to represent these shapes. The

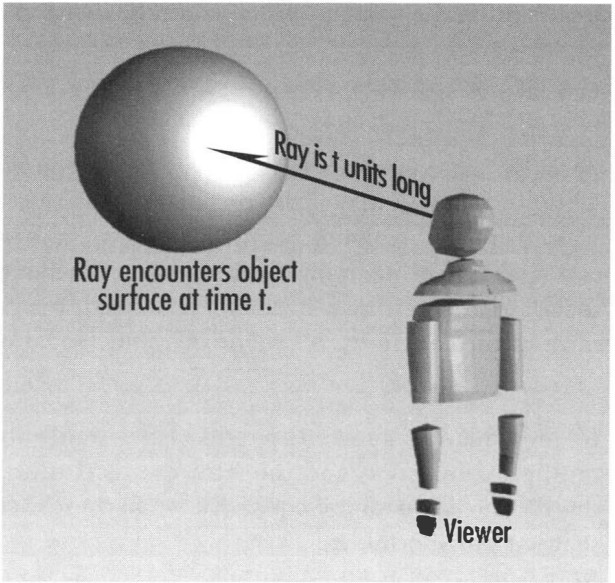


Figure 8-10 At time *t*, the ray of light has traveled *t* units

ray of light itself is measured by a unit called *t*, which stands for time. Since the ray is presumably moving, albeit backward, it will reach different points in the scene at different times. The value *t* represents the amount of time that the ray requires to travel *t* coordinate units through the scene. If *t* is equal to, say, 10, then the ray has traveled 10 coordinate units (see Figure 8-10). Thus, if we know that the ray encounters a surface at time *t*, we can calculate the actual coordinate position of that surface in space with these equations:

```
x = xd * t + x0
y = yd * t + y0
z = zd * t + z0
```

These equations are said to be in parameterized form, because the value of the equations depends on the value of the variable *t*. Of course, this still doesn't tell us whether the ray encounters any objects along the way. To determine that, we must combine the parameterized equation of the ray with the equation of a sphere or an infinite plane. The equation of a sphere with its center at coordinates *l,m,n* and radius *r* looks like this:

```
l2 + m2 + n2 - r2 = 0
```

where *l*, *m*, and *n* are the three-dimensional coordinates of the sphere's center and *r* is its radius (all information that we've stored in the *object_type* structure

representing the sphere). When combined with the parameterized equation of the ray, this becomes:

```
(xd^2 + yd^2 + zd^2) * t^2 + 2 *(x0*xd + y0*yd +z0*zd) * t + (x0^2 + y0^2 +
z0^2) - 1 = 0
```

So what are we supposed to do with this equation? If you remember your high school math, you may recognize that the equation above is a quadratic equation. This is quite fortunate, because there's a standard formula — known, sensibly enough, as the quadratic formula — for solving quadratic equations. Actually, there are *two* formulas, because every quadratic has two solutions, known as the roots of the equations. A generic quadratic equation looks like this:

```
a*x^2 + b*x + c = 0
```

In the equation above, the term that is multiplied by t^2 is *a,* the term that is multiplied by *t* is *b,* and the term that isn't multiplied by anything is *c*. To find the roots of a quadratic equation, we first must calculate its *discriminant.* We do that with this formula:

```
discriminant =  b^2 - 4*a*c
```

Once we have the discriminant, we can calculate one of the two roots with this formula:

```
root1 = (-b + sqrt(discriminant)) / 2*a
```

We need only change one operator in that formula to calculate the other root:

```
root2 = (-b-sqrt(discriminant)) / 2*a
```

The discriminant and the root of the parameterized equation of the ray and sphere tell us all we need to know about those two entities. If the discriminant is equal to or less than 0, the ray never intersects the sphere. If the discriminant is positive, then the smaller of the two roots represents the time *t* at which the ray strikes the sphere. (If the sphere is transparent or semitransparent, the larger of the two roots would represent the time *t* at which the ray passed through the opposite side of the sphere.)

We can translate the time *t* into the actual x, y, and z coordinates in space where the ray strikes the sphere using the equations we saw earlier in this chapter. Remember that time *t* represents the time the ray takes to traverse *t* coordinate units. The change in the x, y, and z coordinates along one unit of the ray is contained in the direction vector *rayvec.* Thus we need only multiply the x, y, and z portions of the direction vector by *t* and add the result to the starting coordinates of the ray to find the position at which the ray strikes the sphere:

```
loc.x=raystart.x+rayvec.x*t
loc.y=raystart.y+rayvec.y*t
loc.z=raystart.z+rayvec.z*t
```

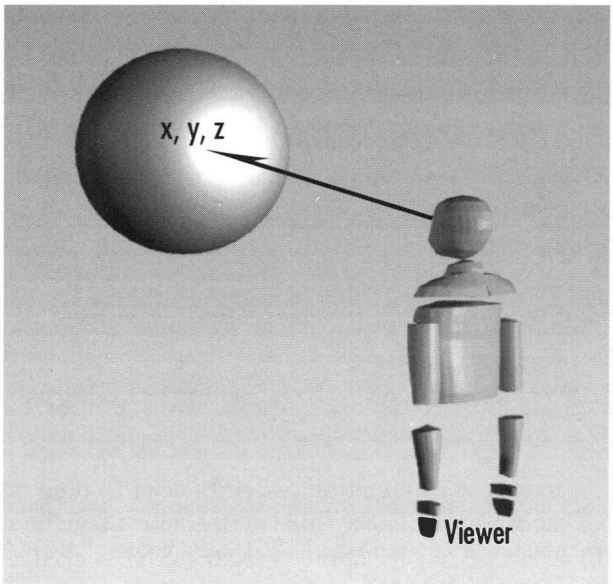


Figure 8-11 The coordinates at which the ray strikes the surface of the sphere

All that remains to do now is to find the angle between the surface of the sphere at the point of intersection and the rays coming from the light source, so that we can adjust the color of the sphere's surface and find out what color to paint the pixel that the ray passes through. To find the vector of a line normal to — that is, perpendicular to — the surface of the sphere at the point where the ray strikes it, we must subtract the coordinates of the center of the sphere (which we'll call *l*, *m*, and *n*) from the coordinates of the point at which the ray strikes its surface (which we'll call *x*, *y*, and *z*) and divide the result by the radius of the sphere (which we'll call *r*):

```
normal.x = (x - l)/r
normal.y = (y - m)/r
normal.z = (z - n)/r
```

This gives us the coordinates in space at which the ray hits the sphere (see Figure 8-11). Obtaining the actual angle between the normal at the sphere's surface and a ray of light coming from the light source is a bit more complicated. We'll talk more about the details of this sort of lightsourcing in a later chapter. For now, we'll simply provide a function called *lightsource()* that will take as parameters the gray-scale color of the surface, the location of a point on that

surface, and the normal of the surface at that point, and return the lightsourced value of the color at that point, which will then be used as the color of the pixel.

Finding an intersection between the ray and an infinite plane is actually a little simpler, which is certainly good news after all that math. The equation for a plane is:

```
ax + by + cz + d = 0
```

where x, y, and z are the coordinates of any point on the plane's surface; a, b, and c are a direction vector of a line normal to — that is, perpendicular to — the plane's surface; and d is the distance along that line from the origin of the universe to the plane.

When combined with the parameterized equation of the line, this becomes:

```
t = -(a*x0 + b*y0 + c*z0 + d) / (a*xd + b*yd + c*zd)
```

To solve this equation, we only need to plug in the appropriate values in place of the variables and we'll have the time *t* at which the ray strikes the plane. If the first part of this equation — a*x0 + b*y0 + c*z0 — is 0, then the ray is parallel to the plane and never strikes it. If it is non-zero and the value of *t* is positive, the ray intersects the plane. (It also intersects the plane if the value is negative, but it intersects it *behind* the viewer, where it cannot be seen.) We can then determine the angle between the plane at that point and the light source exactly as we did with the sphere, except that the normal vector for the plane is already known. (It's stored in the *object_type* variable representing the plane.)

Now that we've covered the relevant math, let's write the *trace_ray()* function.

A Ray-Tracing Function

Here's the prototype for the function:

```
int trace_ray(vector_type rayvec,vector_type raystart)
```

We'll begin the function by declaring a few relevant variables:

```
int color;                                  // The color of the ray
double a,b,c,d,x1,y1,z1,x,y,z;              // Temporary values
double t,t0,t1,root1,root2,discriminant;    // Temporary values
double X0,Y0,Z0,Xd,Yd,Zd,dot,dot2;          // Temporary values
double prev_t=MAXDOUBLE;                    // Previous intersection
vector_type norm,loc,plane_loc,pixel,diff,temp,plane; // Temp vectors
```

Most of these will be temporary placeholders for values already contained either in the parameters passed to the function or in the *object_type* array. The double-precision variable *prev_t* is set to *MAXDOUBLE*, the largest double-

precision variable that the compiler can deal with without losing precision, for reasons that we'll explore in a moment.

The variable *color* will hold the gray-scale color that we'll be returning to the calling function. Initially, we'll set it to the background color:

```
color=BACKGROUND_COLOR;
```

If no intersections are found between the ray and an object, this value will be passed unchanged to the calling function.

The rest of the function will be structured as a *for()* loop that will iterate through all of the objects in our ray-traced universe:

```
for (int ob=0; ob<NUMBER_OF_OBJECTS; ob++) {
```

Inside this loop we're going to place a large *switch()* statement, in which each *case* will handle one of the types of objects that we've defined in our world structure. Since only two types of objects have been defined — spheres and infinite planes — only two *cases* will be necessary:

```
switch (objects[ob].type_of_object) {
```

The *type_of_object* field tells us what type of object each element of the *objects* array represents. For our first *case,* we'll check to see if it's an infinite plane:

```
case INFINITE_PLANE:
```

If it *is* an infinite plane, we'll set the *vector_type* variable *norm* equal to the plane's surface normal:

```
norm=objects[ob].obj.plane->surface_normal;
```

We'll further set the variables *a, b, c,* and *d* equal to the x, y, and z elements of the normal and the distance between the plane and the origin:

```
a=norm.x;
b=norm.y;
c=norm.z;
d=objects[ob].obj.plane->distance;
```

And we'll set the variables *X0, Y0,* and *Z0* to the starting coordinates of the ray and the variables *Xd, Yd,* and *Zd* to the direction vector of the ray:

```
X0=raystart.x;
Y0=raystart.y;
Z0=raystart.z;
Xd=rayvec.x;
Yd=rayvec.y;
Zd=rayvec.z;
```

We can then calculate the first portion of the combined equation of the line and plane:

```
double vd=a*Xd+b*Yd+c*Zd;
```

If this isn't equal to 0, we'll continue:

```
if (vd!=0) {
```

Then we'll calculate the second part of the equation:

```
double v0=-(a*X0+b*Y0+c*Z0+d);
```

and derive the value of *t:*

```
t=v0/vd;
```

This is where the variable *prev_t* comes in. If the ray intersects more than one surface in the scene, we only want to know the color of the nearest one, since it will block the ray and keep it from striking the second object (unless the first surface is transparent, a possibility that we are not including in our ray tracer). The variable *prev_t* holds the distance to the last surface the ray struck. (Initially, we set it to the farthest distance possible, so that any objects that it struck would be at smaller distances.) We check to see if *t* is both positive (so that we know the surface is in front of the viewer) and smaller than *prev_t* before we continue:

```
if ((t<prev_t)&&(t>=0)) {
```

If *t* passes both tests, we set *prev_t* equal to the distance to this surface:

```
prev_t = t;
```

and calculate the coordinates in space of the intersection point, storing them in the *vector_type* variable *loc:*

```
loc.x=X0+Xd*t;
loc.y=Y0+Yd*t;
loc.z=Z0+Zd*t;
```

To get the actual lightsourced color of the surface, we pass the color of the plane (stored in *objects[ob].color)* to the *lightsource()* function:

```
color=lightsource(objects[ob].color,loc,norm);
```

And that's all we need to do if the object is a plane. At the end of the function, we'll return the value of *color* to the calling function.

Tracing a Sphere

We follow a similar course of action if the object is a sphere:

```
case SPHERE:
```

We set the variables *Xc, Yc,* and *Zc* equal to the coordinates of the center of the sphere, and *Sr* equal to the radius:

```
double Xc=objects[ob].obj.sphere->l;
double Yc=objects[ob].obj.sphere->m;
double Zc=objects[ob].obj.sphere->n;
double Sr=objects[ob].obj.sphere->r;
```

Once again, we set *X0, Y0,* and *Z0* equal to the starting point of the ray, and *Xd, Yd,* and *Zd* to the direction vector of the ray:

```
X0=raystart.x;
Y0=raystart.y;
Z0=raystart.z;
Xd=rayvec.x;
Yd=rayvec.y;
Zd=rayvec.z;
```

Then we set the variables *b* and *c* equal to the b and c components of the quadratic equation:

```
b=2.0*(Xd*(X0-Xc)+Yd*(Y0-Yc)+Zd*(Z0-Zc));
c=(X0-Xc)*(X0-Xc)+(Y0-Yc)*(Y0-Yc)+(Z0-Zc)*(Z0-Zc)-Sr*Sr;
```

We then calculate the discriminant using the quadratic formula and check to see if it's either 0 or positive:

```
if ((discriminant=b*b-4*c)>=0) {
```

If it is, we calculate the first and second roots of the equation:

```
double sd=sqrt(discriminant);
t0=(-b-sd)/2;
t1=(-b+sd)/2;
```

We then set *t* equal to the smaller of *t0* and *t1:*

```
if ((t0>0)||(t1>0)) {
  if (t0>0) t=t0;
  if ((t1<t0)&&(t1>=0)) t=t1;
```

If *t* is less than *prev_t,* we calculate the position of the point in space where the ray intersects the sphere and the surface normal of the sphere at that point:

```
if (t<prev_t) {
  prev_t=t;

  // Calculate the coordinates of intersection:
  loc.x=X0+Xd*t;
  loc.y=Y0+Yd*t;
  loc.z=Z0+Zd*t;

  // Find the normal vector at that point:
  norm.x=(loc.x-Xc)/Sr;
  norm.y=(loc.y-Yc)/Sr;
  norm.z=(loc.z-Zc)/Sr;
```

Then we call the *lightsource()* function to get the color of the surface at that point:

```
color=lightsource(objects[ob].color,loc,norm);
```

And that's it. The only thing left to do in the *trace_ray()* function is to terminate the loops and *if()* and return the value of *color* to the calling function.

The lightsource() Function

We won't discuss the *lightsource()* function in detail, because we'll be covering lightsourcing in a later chapter. The prototype looks like this:

```
int lightsource(int color,vector_type loc,vector_type norm)
{
```

We begin by defining some variables:

```
double newcolor;
vector_type lightvec;
double integer;
```

To obtain a vector from the surface to the light source, we subtract the surface location from the coordinates of the light source:

```
lightvec.x=LIGHT_SOURCE.x-loc.x;
lightvec.y=LIGHT_SOURCE.y-loc.y;
lightvec.z=LIGHT_SOURCE.z-loc.z;
```

Then we calculate the distance to the light source using the Pythagorean formula:

```
long distance=sqrt(lightvec.x*lightvec.x+lightvec.y*lightvec.y
                   +lightvec.z*lightvec.z);
```

If the distance is 0, we fudge it up to 1 to avoid divide-by-0 errors:

```
if (distance==0) distance=1;
```

Then we use the distance to find the unit vector to the light source:

```
double xlight=(double)lightvec.x/distance;
double ylight=(double)lightvec.y/distance;
double zlight=(double)lightvec.z/distance;
```

The standard method for calculating the intensity of light on a surface is to multipy the normal vector of the surface by the unit vector to the light source:

```
double intensity=norm.x*xlight+norm.y*ylight+norm.z*zlight;
```

This will give us a number between 0 and 1 representing the intensity of the light. We add the ambient intensity to this value:

```
intensity+=AMBIENT_INTENSITY;
```

If the result is greater than 1, we round it off to 1:

```
if (intensity>1.0) intensity=1.0;
```

And we multiply the resulting intensity by the gray-scale color of the surface:

```
newcolor=color*intensity;
```

We don't want the edge between shades of gray in the final image to be too sharp, since this can create a phenomenon known as "banding." Since the gray-scale value obtained from the previous equation probably has a fractional value, we flip a virtual coin with the *random()* function to decide whether we should round that value up to the next integer with the *ceil()* function or down to the previous integer with the *floor()* function. This technique is called dithering, and will give a randomly speckled look to the shading in the final image:

```
if (modf(newcolor,&integer) * 10000 > random(10000)) newcolor=ceil(newcolor);
else newcolor=floor(newcolor);
```

It's possible that we just rounded a gray-scale value above the range allowed by our program, so we check for values greater than 63 and round them down to 63:

```
if (newcolor>63) newcolor=63;
```

Finally, we pass the lightsourced color back to the calling function:

```
  return (int) newcolor;
}
```

The plot() Function

The *plot()* function is so simple that it's barely worth describing. We use the same formula that we've used in earlier chapters for calculating a video memory offset from a pair of coordinates:

```
void plot(int x,int y,int color)
{
  int c;

  screen[y*320+x]=color;
}
```

The RTDEMO Program

The complete text of the ray-tracing program, called RTDEMO.CPP, is in Listing 8-1:

Listing 8-1 The RTDEMO.CPP program

```
// RTDEMO.CPP Version 1.0
// Simple demonstration of ray tracing (in gray scale)
//
// Written by Christopher Lampton
//   for Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  "screen.h"
#include  "pcx.h"
#include  "math.h"
#include  "time.h"
#include  "values.h"

const RECTANGLE=1;
const INFINITE_PLANE=2;
const SPHERE=3;

struct vector_type {
  double x,y,z;
};

struct plane_type {
  vector_type surface_normal;
  double distance;
};

struct sphere_type {
  double l,m,n; // Center of sphere
  double r;     // Radius of sphere
};

union object_union {
  plane_type *plane;
  sphere_type *sphere;
};

struct object_type {
  int type_of_object;
  int color;
  object_union obj;
};

const NUMBER_OF_OBJECTS=2;
const BACKGROUND_COLOR=6;
```

```
const double AMBIENT_INTENSITY=.14;
const vector_type LIGHT_SOURCE={500,-200,150};
const VIEWER_X=160;
const VIEWER_Y=100;
const VIEWER_DISTANCE=100;

struct plane_type plane={ {0,-1,0},       // Surface normal
                          500             // Distance from origin
};
struct sphere_type sphere={0,40,300,   // Coordinates of center
                          100}         // Radius
;

struct object_type objects[NUMBER_OF_OBJECTS]={
  {SPHERE,56,(plane_type *) &sphere},
  {INFINITE_PLANE,48,(plane_type *) &plane}
};

int trace_ray(vector_type rayvec,vector_type raystart);
void plot(int x,int y,int color);
vector_type getloc(double t,vector_type rayvec,vector_type raystart);
int lightsource(int color,vector_type loc,vector_type norm);

int image=13;
char far *screen;

void main()
{
  char pal[768];
  vector_type rayvec,raystart;

  randomize();

  for (int grey=0; grey<64; grey++) {
    pal[grey*3] = (int)grey;
    pal[grey*3+1] = (int)grey;
    pal[grey*3+2] = (int)grey;
  }

  int oldmode=*(int *)MK_FP(0x40,0x49);  // Save previous
                                         //  video mode
  screen=(char far *)MK_FP(0xa000,0);
  setmode(0x13);                         // Set mode 13h
  setpalette(pal,0,256);

  cls(screen);

  for (int yscreen=0; yscreen<200; yscreen++) {
    for (int xscreen=0; xscreen<320; xscreen++) {
      raystart.x=xscreen-VIEWER_X;
      raystart.y=(yscreen-VIEWER_Y)*1.2;
      raystart.z=VIEWER_DISTANCE;
```

continued on next page

continued from previous page

```
      double distance=sqrt(raystart.x*raystart.x+raystart.y*raystart.y
                           +raystart.z*raystart.z);
      rayvec.x = raystart.x/distance;
      rayvec.y = raystart.y/distance;
      rayvec.z = raystart.z/distance;
      plot(xscreen,yscreen,trace_ray(rayvec,raystart));
    }
  }

  // Wait for user to hit a key:

  while (!kbhit());

  setmode(oldmode); // Reset video and exit
}

int trace_ray(vector_type rayvec,vector_type raystart)
{
  int color;
  double a,b,c,d,x1,y1,z1,x,y,z;
  double t,t0,t1,root1,root2,discriminant;
  double X0,Y0,Z0,Xd,Yd,Zd,dot,dot2;
  double prev_t=MAXDOUBLE;
  vector_type norm,loc,plane_loc,pixel,diff,temp,plane;

  color=BACKGROUND_COLOR;
  for (int ob=0; ob<NUMBER_OF_OBJECTS; ob++) {
    switch (objects[ob].type_of_object) {
      case INFINITE_PLANE:
        norm=objects[ob].obj.plane->surface_normal;
        a=norm.x;
        b=norm.y;
        c=norm.z;
        d=objects[ob].obj.plane->distance;
        X0=raystart.x;
        Y0=raystart.y;
        Z0=raystart.z;
        Xd=rayvec.x;
        Yd=rayvec.y;
        Zd=rayvec.z;
        double vd=a*Xd+b*Yd+c*Zd;
        if (vd!=0) {
          double v0=-(a*X0+b*Y0+c*Z0+d);
          t=v0/vd;
          if ((t<prev_t)&&(t>=0)) {
            prev_t = t;
            loc.x=X0+Xd*t;
            loc.y=Y0+Yd*t;
            loc.z=Z0+Zd*t;
            color=lightsource(objects[ob].color,loc,norm);
          }
```

```
        break;
      case SPHERE:
        double Xc=objects[ob].obj.sphere->l;
        double Yc=objects[ob].obj.sphere->m;
        double Zc=objects[ob].obj.sphere->n;
        double Sr=objects[ob].obj.sphere->r;
        X0=raystart.x;
        Y0=raystart.y;
        Z0=raystart.z;
        Xd=rayvec.x;
        Yd=rayvec.y;
        Zd=rayvec.z;
        b=2.0*(Xd*(X0-Xc)+Yd*(Y0-Yc)+Zd*(Z0-Zc));
        c=(X0-Xc)*(X0-Xc)+(Y0-Yc)*(Y0-Yc)+(Z0-Zc)*(Z0-Zc)-Sr*Sr;
        if ((discriminant=b*b-4*c)>=0) {
          double sd=sqrt(discriminant);
          t0=(-b-sd)/2;
          t1=(-b+sd)/2;
          if ((t0>0)||(t1>0)) {
            if (t0>0) t=t0;
            if ((t1<t0)&&(t1>=0)) t=t1;
            if (t<prev_t) {
              prev_t=t;
              loc.x=X0+Xd*t;
              loc.y=Y0+Yd*t;
              loc.z=Z0+Zd*t;
              norm.x=(loc.x-Xc)/Sr;
              norm.y=(loc.y-Yc)/Sr;
              norm.z=(loc.z-Zc)/Sr;
              color=lightsource(objects[ob].color,loc,norm);
            }
          }
        }
        break;
    }
  }
  return color;
}

void plot(int x,int y,int color)
{
  int c;

  screen[y*320+x]=color;
}

int lightsource(int color,vector_type loc,vector_type norm)
{
  double newcolor;
  vector_type lightvec;
  double integer;
```

continued on next page

continued from previous page

```
  lightvec.x=LIGHT_SOURCE.x-loc.x;
  lightvec.y=LIGHT_SOURCE.y-loc.y;
  lightvec.z=LIGHT_SOURCE.z-loc.z;
  long distance=sqrt(lightvec.x*lightvec.x+lightvec.y*lightvec.y
                     +lightvec.z*lightvec.z);
  if (distance==0) distance=1;
  double xlight=(double)lightvec.x/distance;
  double ylight=(double)lightvec.y/distance;
  double zlight=(double)lightvec.z/distance;
  double intensity=norm.x*xlight+norm.y*ylight+norm.z*zlight;
  if (intensity<0.0) intensity=0;
  intensity+=AMBIENT_INTENSITY;
  if (intensity>1.0) intensity=1.0;
  newcolor=color*intensity;
  if (modf(newcolor,&integer) * 10000 > random(10000)) newcolor=ceil(newcolor);
  else newcolor=floor(newcolor);
  if (newcolor>63) newcolor=63;
  return (int) newcolor;
}
```

To run the program, go to the directory called RAYTRACE and type RTDEMO. The image of a ray-traced gray sphere and plane will appear (see Figure 8-12).

A second version of the program is available in that directory under the name RAY2. Type RAY2 and you'll see a second trace, with a larger number of spheres in several different shades of gray (see Figure 8-13). If you have a copy of the Borland C++ compiler, you might want to design some scenes of your own and recompile the tracer.

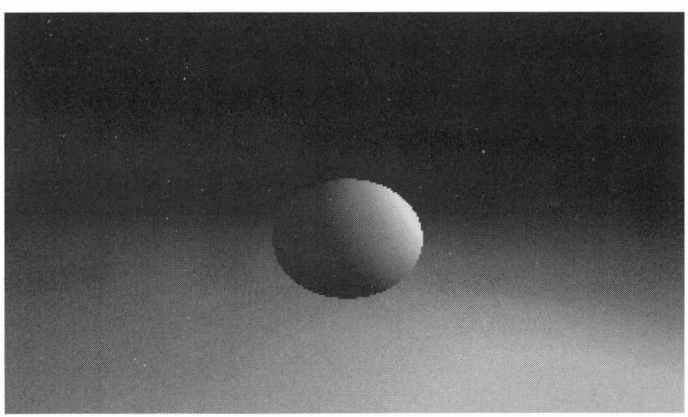


Figure 8-12 The ray-traced sphere and plane

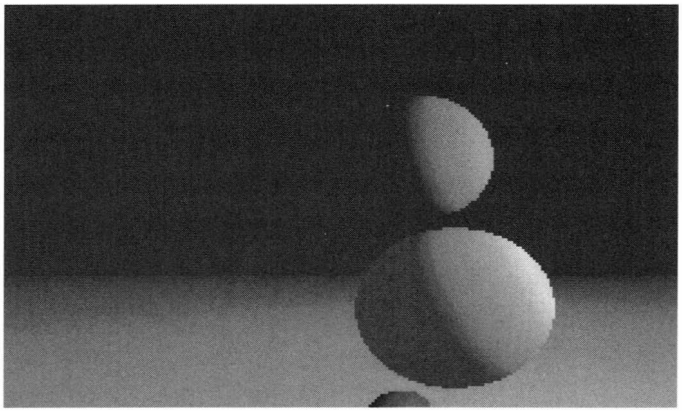


Figure 8-13 Ray-traced multiple spheres

Extending the Ray Tracer

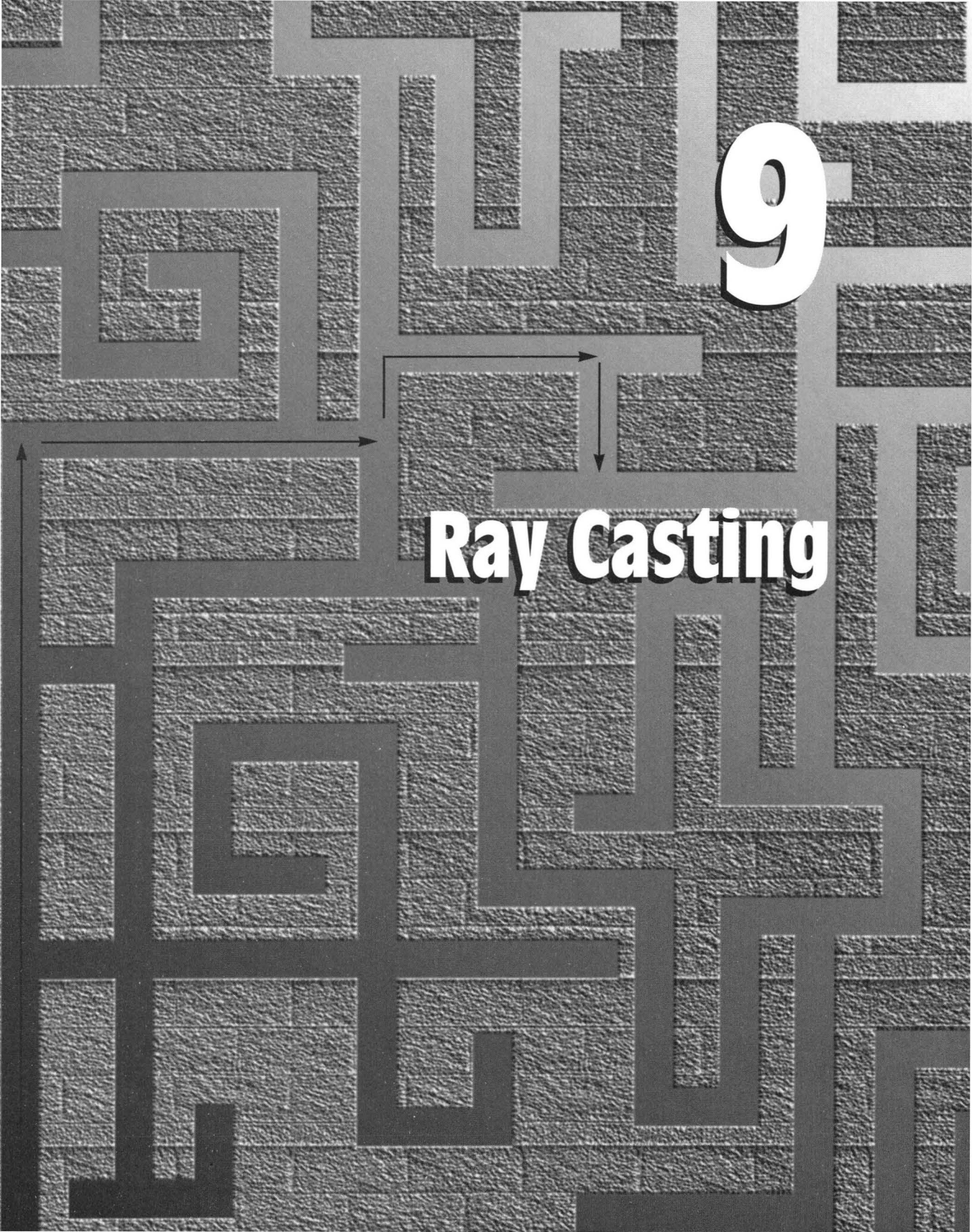
If you're interested in pursuing the subject of ray tracing at greater lengths, there are a number of ways in which this ray tracer could be extended. Two of the most interesting extensions would be to add texture mapping and reflective surfaces. Then you could create one of the favorite images of the novice ray tracer: a reflective sphere hovering over an infinite checkerboard.

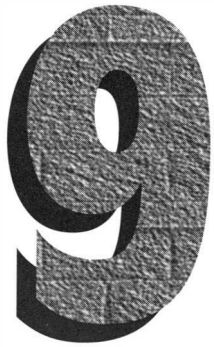
You could also add additional primitive shapes, including polygons and toruses, out of which any number of complex images could be fashioned. If you're feeling really ambitious, you could add full-color capability to the ray tracer. If you have a graphics card capable of displaying 24-bit color (a common feature of most recent graphics cards), you could even create 16-million-color pictures, with delicate shading and vibrantly colored surfaces. Of course, if you don't feel up to writing a full-fledged ray tracer, you can always use one of the freeware and shareware ray tracers such as POVRAY, POLYRAY, or VIVID.

Ray-Traced Mazes?

So are we going to use ray-tracing techniques to create a maze game? Not exactly. As you may have noticed while using this ray tracer, it can take from several seconds (on fast machines) to several minutes (on slower machines) to trace a complete image. If we were to buckle down and optimize this program, we could make it considerably faster — there are a number of optimization techniques that can be used on ray tracers — but it still wouldn't be able to maintain the sort of frame rate required of a maze game.

Instead, we are going to take some of the principles used in this ray tracer and put together a brand-new technique for generating maze images, a technique that can be used to maintain quite a respectable frame rate. The technique that we'll be developing is sometimes referred to as *ray casting*.


9
Ray Casting



A ray tracer creates its miraculously realistic renderings by tracing an imaginary ray of light through every pixel in an image. That's a lot of rays. Even when creating a relatively low-resolution mode 13h full-screen image, where there are 320 pixels in each of 200 rows, a ray tracer must trace 64,000 individual rays. That takes a lot of time.

In a naive ray tracer such as the one we developed in the last chapter, each of these rays must be compared with every primitive object in the scene, which is even more time consuming. Industrial-strength commercial ray tracers use clever algorithms to avoid comparing the ray with objects far from its path, but the number of comparisons between the ray and the objects in the scene is still usually quite large.

If we are to come up with some sort of variant on ray tracing that can operate at speeds sufficient for real-time animation, we'll need to put the bulk of our optimizations into reducing the number of rays to be traced and reducing the number of comparisons that need to be made between rays and objects. With that goal in mind, let's do some brainstorming.

Reducing the Number of Comparisons

To reduce the number of comparisons between rays and objects, it would be useful if we had a clear idea in advance about the sort of world the rays will be traveling through. A ray tracer can trace just about any kind of scene that a computer artist wants it to trace, but we don't necessarily need that versatile a renderer. In earlier chapters we've worked with a much more limited universe, one that consists of a floor, a ceiling, and a lot of cubes placed at fixed intervals to form the walls of a maze. So in this chapter we'll develop a renderer that can only work within such a universe.

Reducing the Number of Rays

We also want to reduce the number of rays to be traced, so we'll place an even more radical limitation on our renderer. Instead of tracing a ray for every pixel in the display, we'll trace a ray for every vertical *column* of pixels in the display. In a full-screen-mode 13h image, this will reduce the number of rays to be traced by a factor of 200!

That may sound like an impossible limitation, but it really isn't. In chapter 6, for instance, we created texture-mapped polygons by drawing vertical slices of the texture one after another from the left side of the polygon to the right. The renderer that we create in this chapter will texture map the interior of an entire maze in much the same way: by dividing it up into vertical slices and casting a

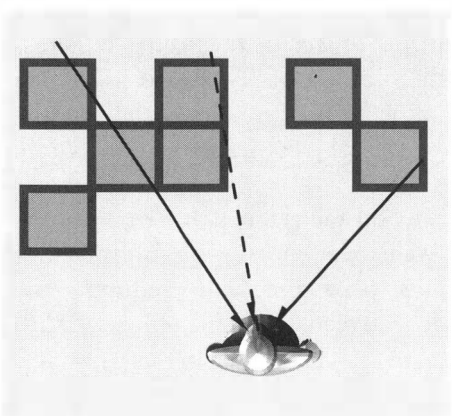


Figure 9-1a An adventurer in the maze. The arrows are the rays of light traveling from the scene to the adventurer's eyes

ray along each slice to determine what textures lie along its path. The simplified architecture that we'll be using, with walls laid down at right angles to one another, makes these simplified rendering methods possible.

Casting Rays

To see how this works, take a look at Figure 9-1a. It depicts an intrepid adventurer standing in the middle of a maze, as viewed from directly above. (Since all you can see is the top of the adventurer's head, you'll have to take my word for it that he or she is intrepid.) The tip of the adventurer's nose tells us in what direction he or she is facing. The arrows zeroing in on the adventurer's face are the rays of light emanating from the maze scene, which form the adventurer's field of view.

Figure 9-1b shows the maze as the adventurer sees it — and as we will be drawing it in the on-screen viewport. The ray at the extreme left in Figure 9-1a corresponds to the column of pixels at the left edge of the viewport in Figure 9-1b. Similarly, the ray at the extreme right in Figure 9-1a corresponds to the column of pixels at the right edge of the viewport in Figure 9-1b. One of the rays of light in Figure 9-1a has been depicted as a dotted line. This ray corresponds to the dotted line drawn over one of the columns near the center of the viewport in Figure 9-1b.

What do I mean when I say that a ray in Figure 9-1a "corresponds" to a column in Figure 9-1b? Imagine that each ray of light is a snake, slithering across

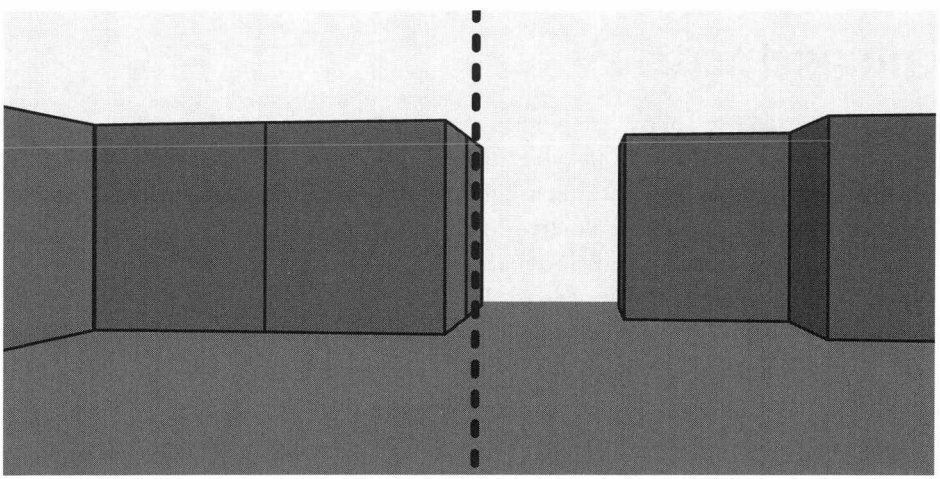


Figure 9-1b The maze as seen by the adventurer

the floor and up and down walls in an arrow-straight line heading directly toward the adventurer's eyes. From the adventurer's point of view, each snake would appear to take a perfectly vertical path along a single column of his or her field of vision. The snake corresponding to the dotted line in Figure 9-1a, for instance, would appear to take a vertical path corresponding to the dotted column in Figure 9-1b.

This means that we can take a single snake-like ray, "cast" it from the adventurer's point of view, and follow it across floors, walls, and ceilings to determine all of the visual information needed to draw an entire column of pixels in the viewport. As in a ray tracer, we'll be following this ray as though it were moving from the adventurer's eyes into the scene, rather than vice versa. If we can develop a speedy enough algorithm for following the path of the ray and determining what lies along it, we should be able to render images of the maze fast enough for real-time animation.

Follow That Ray

We'll break our ray-casting algorithm into two parts. One of these parts will follow the ray across the floors and ceilings of the maze, and the second will follow it along the walls. We'll refer to these two parts whimsically as the *floorcasting* and the *wallcasting* parts. The wallcasting will be the easier of these two parts to implement efficiently, because the geometry of the maze only allows walls to occur at fixed locations, at the edges of the cube-shaped blocks that make up the maze. So we'll implement this part of the algorithm first.

Wallcasting

The way in which we follow the ray as it encounters the walls of our maze is heavily dependent on the way in which information about the maze is stored in the computer's memory. So let's quickly review the maze representation that we've been using throughout this book. In the course of this chapter, we'll expand on this storage system, creating additional arrays of variables to store the information that our ray-casting algorithm needs to draw the walls, floors, and ceilings of the maze.

Until now, we've represented our mazes using a two-dimensional array in which every element corresponds to a section of a maze grid, as depicted in Figure 9-2. A 0 value in an element of the maze array indicates that the equivalent section of the maze grid is empty (except, presumably, for the presence of a floor and a ceiling). A non-zero value in an element of the maze array

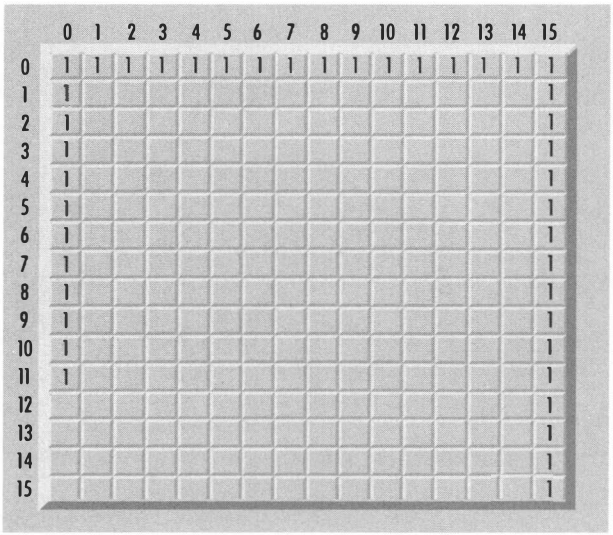


Figure 9-2 The two-dimensional maze grid and the array elements that correspond to the sections of the grid

indicates that the equivalent section of the maze grid contains a solid cube that stretches from floor to ceiling and completely fills the grid section. The four vertical sides of each cube form the walls of the maze.

This maze grid can be looked at as a kind of two-dimensional Cartesian coordinate system. Each grid location is identified by a pair of numbers, which can be thought of as x,y coordinates. We can track a line across this grid of maze squares in the same way that we track lines across the pixel matrix of the mode 13h display. In fact, we could use Bresenham's algorithm to track a ray from maze square to maze square, just as we used it in chapter 2 to draw straight lines made up of pixels.

This would be a fast and efficient way to track rays emanating from the player's position across the grid of maze squares, but it isn't accurate enough for our purposes. It would tell us if the ray encountered a wall; it would even tell us which wall of which maze square the ray encountered. But it wouldn't tell us the precise point on the wall that the ray encountered. As you'll see later in this chapter, we need that information to determine just how to draw the wall.

The problem lies in the coordinate system that we're using to describe the maze grid. It isn't fine enough. We need a coordinate system that will not only allow us to specify maze cube positions throughout the maze grid, but that will allow us to identify specific positions within a maze cube.

To solve this problem, we're going to introduce a second coordinate system that we will use when we need more specific information about the path of a ray.

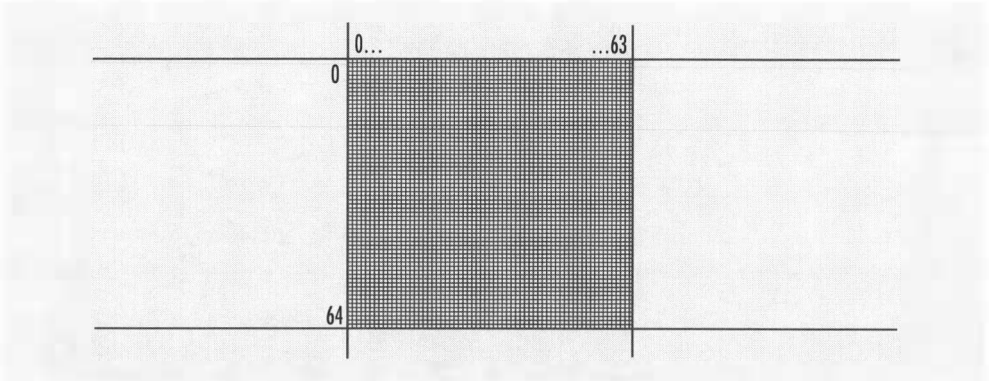


Figure 9-3 The fine coordinate system is nested inside the maze grid, with 64 by 64 coordinates inside each grid square

We'll call this second coordinate system the *fine coordinate system*. So that there won't be any confusion about these coordinate systems, we'll continue referring to the original coordinate system as the *maze grid*.

The Fine Coordinate System

The fine coordinate system will be nested inside the maze grid, as shown in Figure 9-3. Each square of the maze grid will in turn contain a 64 by 64 portion of the fine coordinate system. We haven't chosen this size arbitrarily. The fact that 64 is a power of 2 will simplify some of the calculations when we optimize the ray-casting engine in chapter 12. And later in this chapter we'll be using 64 by 64 bitmaps to map textures onto the walls of the maze cubes. The fact that the pixel matrix of these bitmaps is identical to the fine coordinate system within the maze cubes will simplify the texture mapping considerably.

In the code that we develop in this chapter, we'll use this fine coordinate system in two different ways: to represent positions within a single grid square, and to represent positions within the maze as a whole. At times, we'll specify the position of a point within a grid square by stating that the point falls at coordinates *fine_x* and *fine_y* within the grid square, with the coordinates *grid_x* and *grid_y*. At other times, we'll specify the position of a point within the fine coordinate system of the entire maze. The difference between these two methods of specifying a position should be obvious from context.

Up until now, we've been using maze grids that have measured 16 by 16 — that is, there have been 16 maze squares in each of 16 rows of maze squares, for a total of 256 maze squares, each of which might or might not contain a maze

cube. We'll continue using grids of this size throughout the rest of this book, though the techniques that I'll be describing will work with mazes of other dimensions. (In some cases, the techniques will assume that the maze dimensions correspond to powers of 2.) However, the fine coordinate system nested within all of these squares will measure 64 x 16 by 64 x 16 — or 1024 by 1024. That means there will be a total of more than 1 million fine coordinate points within each maze! Keeping track of all these points is not as difficult as you might think.

From time to time, we'll need to translate back and forth between the fine coordinate system and the maze grid coordinate system. This is actually a simple thing to do. To discover which grid square contains a given pair of x,y coordinates in the fine coordinate system, we'll divide the fine coordinates by 64, like this:

```
grid_x = fine_x / 64;
grid_y = fine_y / 64;
```

Because we've had the foresight to use a power of 2 here, this can, if necessary, be rewritten as a shift operation, which is often faster than a division:

```
grid_x = fine_x >> 6;
grid_y = fine_y >> 6;
```

To find the coordinates of these two points *inside* the grid square at *grid_x, grid_y*, we can take the values of *fine_x,fine_y* modulo 64 (that is, the remainder after these values have been divided by 64), like this:

```
inside_x = fine_x % 64
inside_y = fine_y % 64
```

The modulus operation takes as long as a division. Since we've used a power of 2, we can also obtain the value of these numbers modulo 64 by ANDing them with 63 (hexadecimal 0x3f):

```
inside_x = fine_x & 0x3f
inside_y = fine_y & 0x3f
```

We can translate coordinates in the other direction as well. If we know that a given point is at coordinates *fine_x,fine_y* within grid square *grid_x,grid_y*, we can determine the fine coordinates of the point within the entire maze like this:

```
maze_x = grid_x * 64 + fine_x;
maze_y = grid_y * 64 + fine_y;
```

where *maze_x* and *maze_y* are the fine coordinates within the maze as a whole. We can also rewrite these two instructions using shifts:

```
maze_x = grid_x << 6 + fine_x;
maze_y = grid_y << 6 + fine_y;
```

Rays Within the Grid

When we cast a ray from the viewer's position into the maze, we'll use the fine coordinate system to trace its path. Thus we'll not only know if the ray strikes a wall, but where on the wall it strikes. (Well, we won't know *exactly* where it strikes, but we'll know within a tolerance of 1/64th of the length of the wall, which is close enough.)

If we were writing a full-featured ray tracer, we would need to compare each ray with every cube in the scene to see if it encountered a wall. But our maze representation only allows walls to occur along the edges of a grid square. So we only have to check for ray-wall collisions along those edges. As we trace the ray, we can think of the lines of the maze grid as corresponding to lines in the fine coordinate grid that are evenly divisible by 64. Since walls can only occur at these points in the grid, we need only check for collisions between the ray and lines in the fine coordinate grid that are divisible by 64. We'll refer to these lines as *grid lines*. Those that run parallel to the x axis we'll call *x grid lines*, and those that run parallel to the y axis we'll call *y grid lines*.

A ray of light, as we saw in the last chapter, can be treated as a straight line. In chapter 5, when we introduced code to clip polygons on a display, we saw that the point where a straight line crosses a second line that is parallel to either the x or y axes of the coordinate system could be calculated using the *point-slope equation*. The grid lines in our maze array are all parallel to either the x or the y axes of the maze coordinate system — either the fine coordinate system or the maze grid coordinate system — so we can use the point-slope equation to determine where the ray crosses a grid line. The point-slope equation requires that we first calculate the slope of the ray:

```
slope = (y2 - y1)/(x2 - x1)
```

where *x1,y1* and *x2,y2* are the coordinates of two points along the ray, with *x1,y1* being closer to the viewer than *x2,y2*. These points can belong to either the fine coordinate system or the maze grid coordinate system, though we'll be using the fine coordinate system because of its greater accuracy.

Once we've calculated the slope of the ray, we can determine the x,y coordinates at which the ray crosses a grid line running parallel to the y axis with this formula:

```
x = grid_x;
y = y1 * slope * (grid_x - x1);
```

where *grid_x* is the x coordinate of the grid line. (Since the grid line is running parallel to the y axis of the coordinate system, the x coordinate will be the same at any point on the line.)

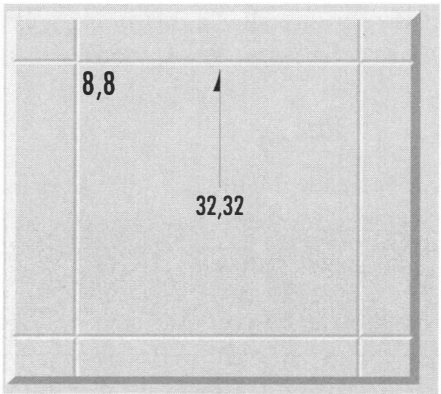


Figure 9-4 Standing at fine coordinates 32,32 within the maze grid square with coordinates 8,8, looking north

We can similarly use the point-slope equation to calculate the x,y coordinates at which the ray crosses a grid line running parallel to the x axis, like this:

```
x = x1 + (grid_ y - y1) / slope;
y = grid_ y;
```

The Basic Ray-Casting Algorithm

To see how this will work, imagine that you are standing at fine coordinates 32,32 within the maze grid square at grid coordinates 8,8, looking due north through the maze (see Figure 9-4). And suppose we want to follow a ray precisely in the center of your field of vision to see if it encounters a wall. First we'll need to derive the fine coordinates within the maze as a whole at which the ray starts. We can do this using the formula for translating from fine coordinates within a grid square to fine coordinates within the maze as a whole:

```
x1 = grid_x * 64 + fine_x;
y1 = grid_y * 64 + fine_y;
```

where x1 and y1 are the fine coordinates at which the ray begins. When we perform this operation on the ray in Figure 9-4, we get starting coordinates of 544,544.

To calculate the slope of the ray, we'll also need the coordinates of a second point further along the ray. We'll arbitrarily use the point that is 1,024 coordinate units distant from the start of the ray. Since the ray is heading straight north (which we'll define as parallel to the y axis of the coordinate system in the

positive y direction), this point will have the same x coordinate as the starting point of the ray, with a y coordinate 1,024 units greater:

```
x2 = x1;
y2 = y1+1024;
```

For the ray in Figure 9-4, this gives us the coordinates 544,1568.

Now we can calculate the slope of the ray:

```
slope = (y2 - y1)/(x2 - x1);
```

It may occur to you that if *x2* and *x1* are the same number, which would be the case if the ray were perfectly vertical (or, in the case of our rays that we'll be casting through the maze, heading due north), this formula will generate a divide-by-0 error. That's a problem with the point-slope equation and with the general mathematical concept of slopes. Mathematicians declare the slope of a vertical line to be infinite, but it's difficult to deal with infinities in C++ (or in any computer language, for that matter). So we'll solve this problem by cheating, as you'll see in a moment.

With that information in hand, we can begin determining whether the ray encounters a wall at some point along its path. Since walls only occur at grid lines, we need to find the next grid line that the ray will cross. Grid lines, you'll recall, are those lines running parallel to the axes of the fine coordinate system that have either an x or a y coordinate that is evenly divisible by 64. The easiest way to determine where the next grid line is would be to find the previous grid line (that is, the one that the ray would have crossed last if it extended from behind the viewer) and add 64 to its x or y coordinate. But how do we find the x or y coordinate of the previous grid line? We can find the y coordinate of the last x grid line that the ray would have passed through by taking the y coordinate of the start of the ray and finding the largest number not exceeding that coordinate that was evenly divisible by 64. Since the y coordinate will be stored in the computer's memory as a binary number, we can do this by simply setting the last six binary digits of the number to 0, which is a matter of ANDing the number with hexadecimal FFC0. Similarly, we can find the x coordinate of the last y grid line crossed by the ray by ANDing the current y coordinate with FFC0.

The ray in Figure 9-4 is heading due north, so it never actually crosses any y coordinate lines (the ones that are parallel to the y axis), since the ray is itself parallel to the y axis. The next x grid line that it crosses has a y coordinate of 578. We can easily calculate this using the formula we just described:

```
grid_x = (x1 && 0xffc0) + 64;
```

We can then find the intersection of the ray with this line using the point-slope equation. If the intersection of the ray and the line occurs at fine

coordinates *xcross,ycross*, we can easily translate these coordinates into maze grid coordinates like this:

```
int xmaze = xcross/64;
int ymaze = ycross/64;
```

Determining whether the ray has struck a wall then becomes a matter of checking the maze array to see if there's a maze cube at those coordinates:

```
if (map[xmaze][ymaze]) {
```

If there is, we then calculate the distance to the wall and draw a vertical segment of the wall in the viewport column position that corresponds to the ray being traced. More about drawing the wall in a moment.

If there isn't a wall at this point, we must cast the ray again from the point of its intersection with the grid line, repeating this process until a wall is encountered or we reach a preset limit to our ray casting.

Rays in All Directions

Not all rays that we cast will head due north, of course. If the ray had been heading due east (in the positive y direction parallel to the x axis), it would intersect the y grid line at 574 rather than the x grid line at 574 (see Figure 9-5). This can also be handled by the algorithm I just described. If the ray heads in the negative direction on the x or y axis, we'll need to look for intersections with the previous grid line rather than the next grid line (see Figure 9-6). Although this might sound like a simple matter of ANDing the current ray coordinates with 0xffc0 and *not* adding 64 to it, the situation is actually a bit more complicated than that. When we divide the resulting coordinates by 64 to derive the maze grid coordinates, we would end up with the coordinates of the square where the ray started rather than the coordinates of the next square it encountered. Thus we must subtract 1 from the x or y coordinate of the x or y grid line to nudge the coordinates into the proper square. This will put us one coordinate off from the actual grid line, but this actually won't cause any problems. The calculation for the next x grid line in the negative direction will look like this:

```
grid_x = (x1 && 0xffc0) -1;
```

The next y grid line can be found in a similar manner.

In practice, we will rarely be dealing with a ray that heads due north, south, east, or west. Most of the rays that we cast won't be parallel with either the x or y axes of the coordinate system. Thus they will cross both x and y grid lines. To determine whether the ray next crosses an x or y grid line, we must determine its intersection with *both* the next x and the next y grid line, determine the distance to each, and check to see which is closer to the current ray coordinates. That will

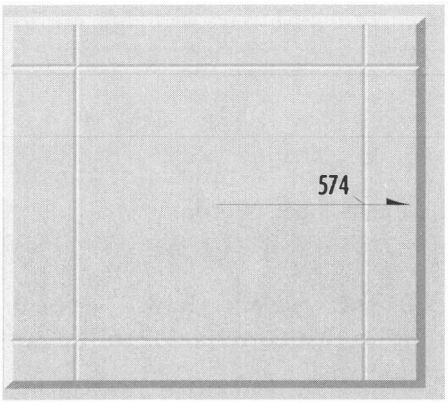


Figure 9-5 A ray heading east intersects the y grid line at coordinate 574

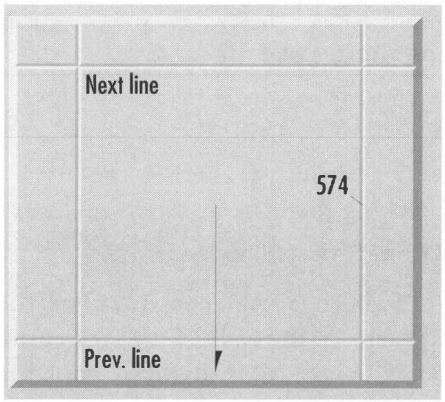


Figure 9-6 Rays heading in the negative x and y direction will strike the next lowest numbered grid line

then become the next point of intersection and we'll check to see if there's a wall at that location.

The Wallcasting draw_maze() Function

Now we're ready to create a ray-casting version of our familiar *draw_maze()* function. Our first version will cast for walls only and will draw those walls in a solid color. Later in this chapter, we'll add texture mapping and floorcasting capabilities to the function. We'll place the first version of this function in the file WALLCAST.CPP.

The prototype for the function, which we'll place in the file WALLCAST.H, looks like this:

```
void draw_maze(map_type map,char far *screen,int xview,
               int yview,float viewing_angle,
               int viewer_height)
```

The first parameter is an array of *map_type*, which will be defined in WALLCAST.H like this:

```
typedef int map_type[16][16];
```

This 16 by 16 array will hold a maze grid of the sort we described earlier, in which each element represents a maze square. Non-zero elements mean that there's a maze cube in the equivalent maze location, zero elements that there's only empty space.

The *screen* parameter is a pointer to video memory or to an off-screen image buffer. The integer variables *xview* and *yview* are the fine coordinates at which the

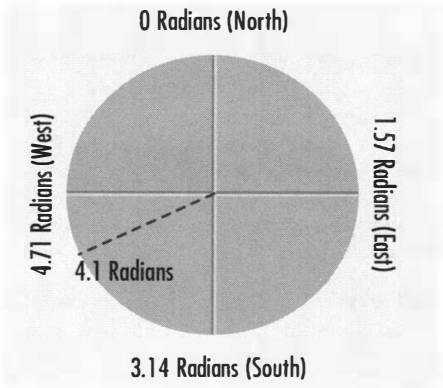


Figure 9-7 An angle of 0 is due north. All other angles are measured relative to this angle

viewer is standing within the maze as a whole. And the floating point variable *viewing_angle* is the angle, in radians, at which the viewer is currently facing, where an angle of 0 is due north. Figure 9-7 shows a "compass" with the approximate radian equivalents for the four cardinal directions and an example of an arbitrary radian direction. (It's impossible to give the exact radian equivalents for the cardinal directions, since radians are based on pi and the equivalent values for 90, 180, and 270 degrees would each have an infinite number of digits after the decimal point.) The *viewer_height* parameter is the desired height of the viewer's eye level above the maze floor, measured in pixels. Since the walls in the maze will be 64 pixels high, we'll generally want to set this value to 32, putting the viewer's eye-level halfway between floor and ceiling. However, other values can be used to create interesting effects, such as crouching, leaping, and standing on top of an object. Values of less than 0 or greater than 63 aren't recommended, since they allow the viewer to see over the top of the walls (revealing holes in the scenery).

We'll place some useful constant definitions at the head of the WALLCAST.CPP file:

```
const WALL_HEIGHT=64;           // Height of wall in pixels
const VIEWER_HEIGHT=32;         // Viewer's eye level in pixels
const VIEWER_DISTANCE=128;      // Viewer distance from screen
const VIEWPORT_LEFT=40;         // Dimensions of viewport
const VIEWPORT_RIGHT=280;
const VIEWPORT_TOP=50;
const VIEWPORT_BOT=150;
const VIEWPORT_HEIGHT=100;
const VIEWPORT_CENTER=100;
```

And we'll declare some necessary variables at the beginning of the function:

```
int sy,offset;           // Pixel y position and offset
float xd,yd;             // Distance to next wall in x and y
int bound_x,bound_ y;    // Coordinates of x and y grid lines
float xcross_x,xcross_ y; // Ray intersection coordinates
float ycross_x,ycross_ y;
int xdist,ydist;         // Distance to x and y grid lines
int xmaze,ymaze;         // Map location of ray collision
int distance;            // Distance to wall along ray
```

We'll see what each of these constants and variables does in the course of the function.

Stepping Through the Columns

The function itself is structured as one large *for()* loop, which will step through all of the vertical columns of pixels from the left side of the viewport to the right:

```
for (int column=VIEWPORT_LEFT; column<VIEWPORT_RIGHT; column++) {
```

This is where things start getting interesting. In order to cast a ray through each column of pixels, we need to know the angle at which the ray is traveling relative to due north (which will have an angle of 0 radians). The viewer is facing at the angle contained in the floating point parameter *viewing_angle,* so let's first calculate the angle relative to this angle. To do this we'll need to use a little trigonometry. Figure 9-8 shows the viewer looking at the video display, as seen from above. There are two lines extending from the viewer's eyes to the screen: Line A represents the straight line distance to the screen (which is contained in the constant VIEWER_DISTANCE), and line C extends from the viewer to an arbitrary column of pixels on the screen. Line A intersects the display directly in the middle, at pixel column 160 on the mode 13h screen. Line C intersects the display at the pixel column contained in integer variable *column,* the index variable for the *for()* loop that we just initialized.

A Touch of Trig

You'll notice that these two lines, combined with the straight surface of the video display, form a triangle. We know three things about this triangle. We know that the side labeled A is VIEWER_DISTANCE in length, we know that the side labeled B is *column-160* in length, and we know that the angle between side A and side B is a right angle — that is, that it measures 90 degrees (or pi/2 radians). What we need to know is the column angle — that is, the angle between sides A and C. There's a simple trigonometric equation that will give us this angle: The angle between A and C equals the arctangent of the length of side B divided by the length of side A. We can easily express this in C/C++, like this:

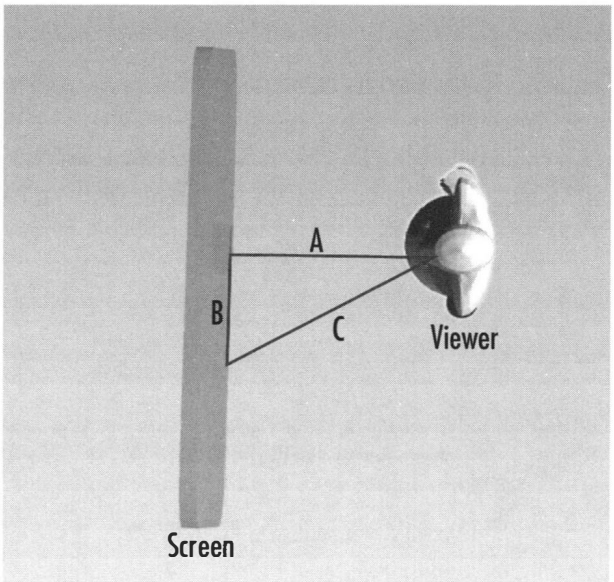


Figure 9-8 A viewer looking at the video display. Line A represents the straight-line distance to the screen and Line C the distance to the column through which the ray is cast. Note that these two lines form a triangle with the screen (Line B)

```
float column_angle=atan((float)(column-160)
                        / VIEWER_DISTANCE);
```

Now we need to find the absolute angle of the ray on the radian compass that I showed you back in Figure 9-7, where 0 radians represents due north. Since we know the angle between the center of the viewer's vision and the ray to the screen column, we can find the absolute angle by adding the column angle to the angle of the center ray, which was passed to the *draw_maze()* function in the parameter *viewing_angle:*

```
float radians=viewing_angle+column_angle;
```

The absolute angle of the ray passing through the column is now contained in the floating point variable *radians.*

Rotating the Ray

In order to compare the ray against the grid lines to see if it encounters a wall, we need to know the fine coordinates of two points on the ray. We already know the

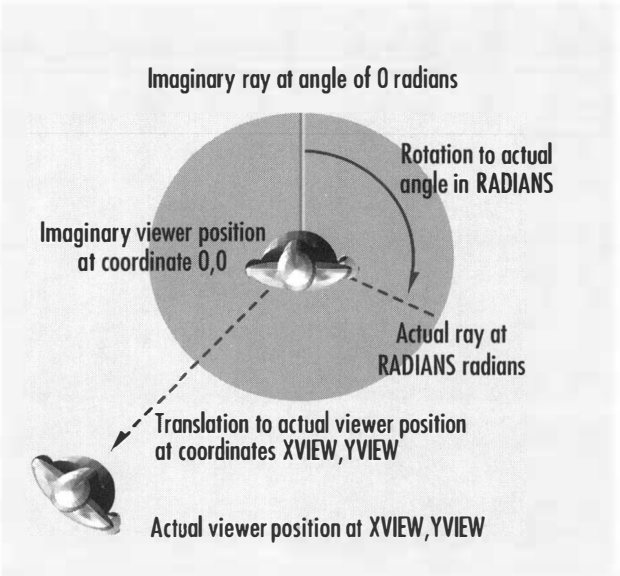

Imaginary ray at angle of 0 radians
Rotation to actual angle in RADIANS
Imaginary viewer position at coordinate 0,0
Actual ray at RADIANS radians
Translation to actual viewer position at coordinates XVIEW,YVIEW
Actual viewer position at XVIEW,YVIEW

Figure 9-9 Rotating and translating an imaginary ray to the viewing angle and view position

fine coordinates of the start of the ray: They are contained in the integer parameters *xview,yview*. But we also need another pair of coordinates from some point further along the length of the ray. Earlier, when we were dealing with a ray that headed due north from the viewer, we simply added 1,024 to the y coordinate at which the ray started. But that only works with a ray traveling at an angle of 0 radians. This ray could be traveling at any possible angle. How do we find a second point along its length?

The answer is that we can use the two-dimensional rotation equation that we introduced in chapter 7. This equation allowed us to rotate a point by an arbitrary number of radians around the 0,0 origin point of a Cartesian coordinate system. As illustrated in Figure 9-9, we'll start with an imaginary line that extends from 0,0 to 0,1024; rotate the second set of coordinates around the first by *radians;* and translate the point relative to the viewer's actual position by adding *xview,yview* to the coordinates. (Translation, you'll remember, is the act of moving a set of coordinates by a given x,y offset to another point in the coordinate system.) This will give us the starting and ending points of a 1,024-pixel-long segment of the actual ray.

To refresh your memory, the two-dimensional rotation equation looks like this:

```
rotated_x = old_x * cos(angle) - old_y * sin(angle)
rotated_y = old_y * cos(angle) - old_x * sin(angle)
```

The *old_x* and *old_y* values here correspond to the x,y coordinates of the endpoint of our imaginary line, which are 0,1024. When we plug those into the equation, we get:

```
rotated_x = 0 * cos(angle) - 1024 * sin(angle)
rotated_y = 1024 * cos(angle) - 0 * sin(angle)
```

Any number multiplied by 0 equals 0, so we can eliminate the terms in this equation that contain 0 factors without affecting the results:

```
rotated_x = 1024 * sin(angle)
rotated_y = 1024 * cos(angle)
```

If we wanted, we could even replace the multiplication with a shift, since we've chosen to multiply by a power of two. But we're not writing optimized code at the moment, so we'll go ahead and use the multiplication in our C++ code for clarity's sake:

```
int x2 = 1024 * (cos(radians));
int y2 = 1024 * (sin(radians));
```

To translate these coordinates to their proper position relative to the viewer, we merely add the viewer's coordinates to them:

```
x2+=xview;
y2+=yview;
```

Casting the Ray

Now we need to start casting the ray. We'll use a pair of floating point variables, *x* and *y*, to hold the coordinates of the ray as we cast and recast it. Initially, these will be set equal to the starting position of the ray, but as we recast the ray from the points at which it intersects the grid lines, we'll update these variables to the coordinates of the intersection:

```
float x=xview;
float y=yview;
```

We're using floating point variables here because we must maintain the precise position at which the ray intersects the grid lines. The fine coordinate system actually isn't fine enough for this purpose; fractional errors would accumulate in the ray coordinates over the course of several encounters, rendering our results inaccurate. This would in turn throw off our calculation of the wall heights, causing odd little "spikes" to appear at the tops and bottoms of the walls. If you're curious about this problem, you can try recompiling this code using integer variables.

(An alternate solution to the problem would be to use an even finer coordinate system nested inside the fine coordinate system, though we would need to use

long integer variables to maintain the ray coordinates in such a system. Yet another alternative, which we'll look at more closely in chapter 12, is to use so-called fixed point math to maintain the coordinate positions. Interestingly enough, these two alternatives would result in almost identical code. Because they're essentially identical, these two methods would be equally fast — and both would be much faster than the floating point code that we're using here.)

The Slope of the Ray

The first step in determining where the ray intersects the grid lines is to find the slope of the ray. We'll break this operation into several steps, first determining the difference in the x and y coordinates from one end of the ray to the other, and storing these values in the variables *xdiff* and *ydiff*:

```
int xdiff=x2-xview;
int ydiff=y2-yview;
```

To determine the slope, we must divide *ydiff* by *xdiff*. If *xdiff* is equal to 0 — meaning that the slope of the line is effectively infinite — this will cause a divide-by-0 error, so we'll need to watch for this condition and take appropriate action if it occurs. We could treat infinite slope as a special case and handle it with special routines, but that's too much of a bother. Instead, we'll cheat:

```
if (xdiff==0) xdiff=1;
```

This way, *xdiff* will never be equal to 0 and the problem will never occur. Of course, this means that we'll never have a ray that's perfectly vertical, but in practice this will scarcely be noticeable.

Now we can obtain the slope of the ray:

```
float slope = (float)ydiff/xdiff;
```

Notice that we force the result of the division to floating point by using a *float* cast on the variable *ydiff*. Otherwise, the result of the division would be an integer and any fractional portion would be truncated *before* being assigned to the floating point variable *slope*.

We'll be using *slope* later as a divisor, so we must cheat a second time to avoid a divide-by-0 error:

```
if (slope==0.0) slope=.0001;
```

Because *slope* is a floating point variable, we can cheat in a more subtle manner than we did with the integer variable *xdiff*, using a very small fraction to represent 0 slope. The result should be unnoticeable.

Casting in a Loop

We'll perform the actual ray casting in a loop. Because we don't know when the loop will end — it will be terminated whenever a wall is found — we'll use an infinite *for()* loop which we can break out of when our work is complete:

```
for (;;) {
```

We could also use a *while()* loop for this purpose, checking on each iteration to see if we've plumbed the depths of the maze up to some predefined limit. However, this would slow down execution somewhat, and isn't strictly necessary if we design the maze so that the ray will *always* encounter a wall. Be warned, though, that if the ray is allowed to pass through the side of the maze and escape — that is, if we fail to surround the perimeter of the maze with solid cubes — this loop will either become infinite or will crash resoundingly.

To find the first collision between the ray and an x grid line, we must determine if the ray is traveling in the positive or negative x direction — that is, toward larger or smaller x coordinate values. This is easy enough to do: If the value of *xdiff* is positive, then the ray is traveling in the positive direction; if *xdiff* is negative, the ray is traveling in the negative direction. If we're traveling in the positive direction, we'll use the formula *x & 0xffc0 + 64* to find the next x grid line:

```
if (xdiff>0) grid_x=((int)x & 0xffc0)+64;
```

If the ray is traveling in the negative direction, we'll use the formula *x & offc0 - 1* to find the previous x grid line minus 1:

```
else grid_x=((int)x & 0xffc0) - 1;
```

We'll repeat this process to find the next (or previous) y grid line:

```
if (ydiff>0) grid_y=((int)y & 0xffc0) +64;
else grid_y=((int)y & 0xffc0) - 1;
```

Calculating the Point of Intersection

The next step is to use the point-slope equation to find the x,y coordinates where the ray crosses the x and y grid lines. We'll use the floating point variables *xcross_x* and *xcross_y* to hold the coordinates at which the ray crosses the x grid line, and the floating point variables *ycross_x* and *ycross_y* to hold the coordinates at which the ray crosses the y grid line:

```
xcross_x=grid_x;
xcross_ y=y+slope*(grid_x-x);
ycross_x=x+(grid_ y-y)/slope;
ycross_ y=grid_ y;
```

Before we can check for the presence of a maze cube, we need to know which of these grid lines the ray strikes first. We'll use the Pythagorean method — adding the squares of the x and y distances to get the straight line distance — for calculating the distance to the x grid line:

```
xd=xcross_x-x;
yd=xcross_y-y;
xdist=sqrt(xd*xd+yd*yd);
```

And we'll use the same method for finding the distance to the y grid line:

```
xd=ycross_x-x;
yd=ycross_y-y;
ydist=sqrt(xd*xd+yd*yd);
```

What we do next depends on which ray is shorter — the one to the x grid line or the one to the y grid line. First we check to see if the x grid line is closer than the y grid line:

```
if (xdist<ydist) {
```

If it is, then we calculate the maze grid coordinates of the maze square that is aligned at that point with the grid line:

```
xmaze=xcross_x/64;
ymaze=xcross_y/64;
```

If there is no maze cube here and we need to start casting again, we must reset the variables *x* and *y* to the coordinates at which the ray intersected the line:

```
x=xcross_x;
y=xcross_y;
```

Even if there is a maze cube at this position, we'll need these coordinates to determine the distance from the viewer to the point of collision.

Finally, we check to see if there's a maze cube at this position. If so, we *break* out of the *for()* loop:

```
  if (map[xmaze][ymaze]) break;
}
```

If the ray to the y grid line is shorter than the ray to the x grid line, we perform the same set of operations on the point of collision with that grid line:

```
  else {
    xmaze=ycross_x/64;
    ymaze=ycross_y/64;
    x=ycross_x;
    y=ycross__y;
    if (map[xmaze][ymaze]) break;
  }
}
```

Drawing the Wall

If the program reaches this point, a cube has been found in the maze square with maze grid coordinates *xmaze,ymaze*. The point of collision between the ray and the maze is at coordinates *x,y*. To find the distance from the viewer to the point of collision, we must subtract the viewer's coordinates from the collision coordinates and apply the Pythagorean distance calculation:

```
xd=x-xview;
yd=y-yview;
distance=(long)sqrt(xd*xd+yd*yd)*cos(column_angle);
```

You'll notice that we multiply the distance to the wall by the cosine of the angle between the center of the viewer's vision and the column through which we've cast the ray. (We earlier stored this angle, in radians, in the variable *column_angle*.) This avoids a "fish-eye" effect in the final image. Figure 9-10 shows why this is so. As the column through which the ray is being cast moves further away from the center of vision, the rays become longer and longer (all other things being equal) and the otherwise straight walls will seem to grow shorter and shorter as they extend outward to each side. This is not, however, the way that the walls would

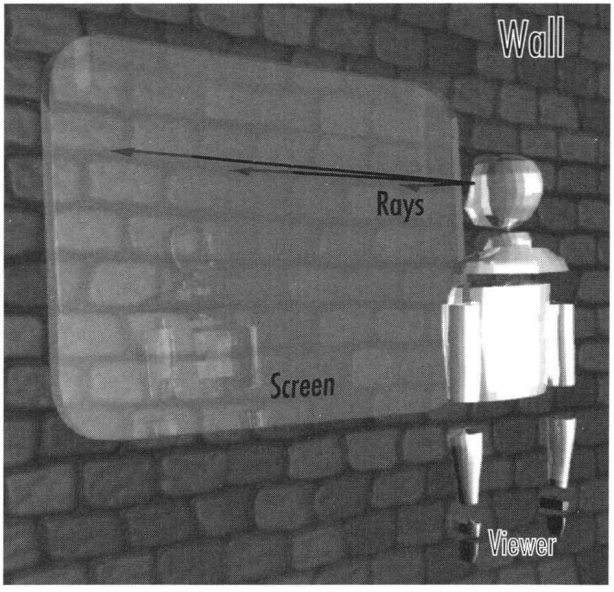


Figure 9-10 The rays grow longer as they move outward from the center of the screen

appear to our eyes. Multiplying the distance by the cosine of the angle provides precisely the adjustment needed to make the walls seem straight and true.

A distance of 0, which can occur if the viewer is smack-dab up against the wall, will cause problems in subsequent calculations, so we cheat once again to avoid the problem:

```
if (distance==0) distance=1;
```

Now we need to know the visible height of the wall, so that we can draw it in the appropriate position in the viewport. We can use the method that we introduced back in chapter 7 for calculating perspective. That method converted the x,y coordinates of a point in space into screen coordinates by dividing them by the z coordinates of the point (which had been previously multiplied by a factor representing the viewer's distance from the screen). We can also use this perspective formula to adjust the wall height, like this:

```
int height = VIEWER_DISTANCE * WALL_HEIGHT / distance;
```

The integer variable *height* now contains the number of visible pixels that are needed to represent the height of the wall in the current screen column. In order to draw this portion of the wall in a solid color, we merely need to draw a column of pixels that tall on the screen in the appropriate position in the viewport. But in order to do that, we need to know at which screen positions the top and bottom pixels on the column will fall. (We could get by with only the top position, but knowing the bottom position simplifies the clipping of the wall column against the top and bottom of the viewport. Later, when we add bitmapped floors to our ray-casting system, we'll need to know where the bottom of the wall is so we can stop drawing the floor when we come to it.)

First we'll calculate the position of the bottommost pixel. If we assume that the viewer's eye level is fixed at a vertical point on the screen defined by the constant VIEWPORT_CENTER (which we've set equal to position 100, directly in the middle of the mode 13h display), then the floor will be at a distance of *viewer_height* below this point. (You'll recall that *viewer_height* is the height of the viewer's eye level relative to the floor of the maze, measured in pixels.) We can adjust *viewer_height* with the perspective equation to get the visible position of the bottom of the wall relative to the center of the viewport, then add VIEWPORT_CENTER to this value to get the equivalent position on the display:

```
int bot = VIEWER_DISTANCE * VIEWER_HEIGHT
          / distance + VIEWPORT_CENTER;
```

The variable *bot* now contains the y coordinate of the bottom of the wall on the display. To get the y coordinate of the top of the wall, we add *height* to this figure:

```
int top = bot - height;
```

Before we can draw this line, we must clip it to the top and bottom of the viewport. If *top* is less than VIEWPORT_TOP — that is, if the column extends above the top of the viewport — we'll subtract the difference between VIEWPORT_TOP and *top* from *height* and set *top* equal to VIEWPORT TOP:

```
if (top < VIEWPORT_TOP) {
  height -= (VIEWPORT_TOP - top);
  top = VIEWPORT_TOP;
}
```

If *bot* is greater than VIEWPORT_BOT — that is, if the column extends below the bottom of the viewport — we'll subtract the difference between *bot* and VIEWPORT_BOT from *height*. (It's not necessary to adjust the value of *bot*, since we won't be using it again.)

The video memory offset of the top pixel in the wall column can be calculated using the standard formula that we've used throughout this book, with *top* representing the y coordinate of the pixel and *column* (which contains the horizontal position of the current viewport column on the display) representing the x coordinate:

```
offset = top * 320 + column;
```

Finally, we'll use a *for()* loop to draw the column, setting the video memory location of the pixel at *offset* to 15 (for the color white) and incrementing the offset position by 320 on each iteration of the loop:

```
    for (int i=0; i<height; i++) {
      screen[offset]=15;
      offset+=320;
    }
  }
}
```

And that's all there is to drawing solid-colored walls using ray-casting techniques. The process that we've just described is repeated for every column of pixels in the viewport, building up a solid-colored rendition of the walls that lie in the viewer's line of sight. The complete *draw_maze()* function, along with related *#includes* and constant declarations, appears in Listing 9-1.

The draw_maze() Function

Listing 9-1 The wallcasting draw_maze() function

```
// WALLCAST.CPP
//
// Function to draw solid-colored walls using ray casting.
// Written by Christopher Lampton for
// Gardens of Imagination (Waite Group Press).
//

#include <stdio.h>
#include <math.h>
#include "raycast.h"
#include "pcx.h"
#include "distance.h"
#include "slope.h"

// Constant definitions:

const WALL_HEIGHT=64;           // Height of wall in pixels
const VIEWER_HEIGHT=32;         // Viewer's eye level in pixels
const VIEWER_DISTANCE=128;      // Viewer distance from screen
const VIEWPORT_LEFT=40;         // Dimensions of viewport
const VIEWPORT_RIGHT=280;
const VIEWPORT_TOP=50;
const VIEWPORT_BOT=150;
const VIEWPORT_HEIGHT=100;
const VIEWPORT_CENTER=100;

extern pcx_struct bitmap;

void draw_maze(map_type map,char far *screen,int xview,
                int yview,float viewing_angle)

// Draws a raycast image in the viewport of the maze represented
// in array MAP[], as seen from position XVIEW, YVIEW by a
// viewer looking at angle VIEWING_ANGLE where angle 0 is due
// north. (Angles are measured in radians.)

{
  // Variable declarations:

  int sy,offset;       // Pixel y position and offset
  float xd,yd;         // Distance to next wall in x and y
  int grid_x,grid_y;   // Coordinates of x and y grid lines
  float xcross_x,xcross_y; // Ray intersection coordinates
  float ycross_x,ycross_y;
```

```
int xdist,ydist;      // Distance to x and y grid lines
int xmaze,ymaze;      // Map location of ray collision
int distance;         // Distance to wall along ray

// ***The raycasting begins:

// Loop through all columns of pixels in viewport:
for (int column=VIEWPORT_LEFT; column<VIEWPORT_RIGHT; column++) {

  // Calculate horizontal angle of ray relative to
  //  center ray:
  float column_angle=atan((float)(column-160)
                      / VIEWER_DISTANCE);

  // Calculate angle of ray relative to maze coordinates
  float radians=viewing_angle+column_angle;

  // Rotate endpoint of ray to viewing angle:
  int x2 = 1024 * (cos(radians));
  int y2 = 1024 * (sin(radians));

  // Translate relative to viewer's position:
  x2+=xview;
  y2+=yview;

  // Initialize ray at viewer's position:
  float x=xview;
  float y=yview;

  // Find difference in x,y coordinates along ray:
  int xdiff=x2-xview;
  int ydiff=y2-yview;

  // Cheat to avoid divide-by-zero error:
  if (xdiff==0) xdiff=1;

  // Get slope of ray:
  float slope = (float)ydiff/xdiff;

  // Cheat (again) to avoid divide-by-zero error:
  if (slope==0.0) slope=.0001;

  // Cast ray from grid line to grid line:
  for (;;) {

    // If ray direction positive in x, get next x grid line:
    if (xdiff>0) grid_x=((int)x & 0xffc0)+64;

    // If ray direction negative in x, get last x grid line:
    else grid_x=((int)x & 0xffc0) - 1;

    // If ray direction positive in y, get next y grid line:
```

continued on next page

continued from previous page

```
      if (ydiff>0) grid_y=((int)y & 0xffc0) +64;

      // If ray direction negative in y, get last y grid line:
      else grid_y=((int)y & 0xffc0) - 1;

      // Get x,y coordinates where ray crosses x grid line:
      xcross_x=grid_x;
      xcross_y=y+slope*(grid_x-x);

      // Get x,y coordinates where ray crosses y grid line:
      ycross_x=x+(grid_y-y)/slope;
      ycross_y=grid_y;

      // Get distance to x grid line:
      xd=xcross_x-x;
      yd=xcross_y-y;
      xdist=sqrt(xd*xd+yd*yd);

      // Get distance to y grid line:
      xd=ycross_x-x;
      yd=ycross_y-y;
      ydist=sqrt(xd*xd+yd*yd);

      // If x grid line is closer...
      if (xdist<ydist) {

        // Calculate maze grid coordinates of square:
        xmaze=xcross_x/64;
        ymaze=xcross_y/64;

        // Set x and y to point of ray intersection:
        x=xcross_x;
        y=xcross_y;

        // Is there a maze cube here? If so, stop looping:
        if (map[xmaze][ymaze]) break;
      }
      else { // If y grid line is closer:

        // Calculate maze grid coordinates of square:
        xmaze=ycross_x/64;
        ymaze=ycross_y/64;

        // Set x and y to point of ray intersection:
        x=ycross_x;
        y=ycross_y;

        // Is there a maze cube here? If so, stop looping:
        if (map[xmaze][ymaze]) break;
      }
    }

    // ***Prepare to draw wall column:
```

```
    // Get distance from viewer to intersection point:
    xd=x-xview;
    yd=y-yview;
    distance=(long)sqrt(xd*xd+yd*yd)*cos(column_angle);
    if (distance==0) distance=1;

    // Calculate visible height of wall:
    int height = VIEWER_DISTANCE * WALL_HEIGHT / distance;

    // Calculate bottom of wall on screen:
    int bot = VIEWER_DISTANCE * VIEWER_HEIGHT
              / distance + VIEWPORT_CENTER;

    // Calculate top of wall on screen:
    int top = bot - height;

    // Clip wall to viewport:
    if (top < VIEWPORT_TOP) {
      height -= (VIEWPORT_TOP - top);
      top = VIEWPORT_TOP;
    }
    if ((top + height) > VIEWPORT_BOT) {
      height -= (bot - VIEWPORT_BOT);
    }

    // ***Draw the wall column:

    // Find video offset of top pixel in wall column:
    offset = top * 320 + column;

    // Loop through all pixels in wall column:
    for (int i=0; i<height; i++) {

      // Set wall pixels to white:
      screen[offset]=15;

      // Advance to next vertical pixel:
      offset+=320;
    }
  }
}
```

The WALLDEMO.CPP Program

To show off the capabilities of this new version of the *draw_maze()* function, we'll write a short program that creates a maze array and calls *draw_maze()* to draw a ray-cast image of the maze. Much of the code in this program will be the same that we've used in earlier programs for setting the video mode, establishing a pointer to video memory, and so on. There are a few new parts as well.

Here's the declaration of the maze array:

```
map_type map={
 {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,1,1,0,1,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1},
 {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
};
```

As noted earlier, zero elements correspond to empty sections of the maze, while non-zero elements correspond to maze cubes. While it may seem wasteful to devote an entire byte of data for a value that can only be zero or non-zero, we'll be using the full value of each byte later in this chapter. Note that, because of the way in which arrays are declared and because we have assumed that north is the positive y direction, the northern end of the maze is the lower part and the southern end is the upper part, the reverse of the standard map convention.

We'll declare a few variables up front to hold the viewing angle, the viewer's height, and the coordinates of the viewer, though only the first of these values will be changing in the remainder of the program:

```
float viewing_angle=0;
int viewer_height=32;
int xview=8*64;
int yview=8*64;
```

In order to make this program slightly interactive, we'll allow the user to pass a command line value representing the viewing angle. Thus we'll need to use the *argc* and *argv* parameters in the declaration of the *main()* function:

```
void main(int argc,char* argv[])
```

If there are no arguments on the command line, we'll default to a viewing angle of 0, which we've already declared. If arguments are detected, we'll convert them to floating point and assign them to the variable *viewing_angle:*

```
if (argc==2) viewing_angle=atof(argv[1]);
```

We'll then proceed to set up mode 13h in the usual fashion. The call to the *draw_maze()* function looks like this:

```
draw_maze(map,screen,xview,yview,viewing_angle,viewer_height);
```

This image will remain on the screen until the user presses a key. The full text of the program WALLDEMO.CPP appears in Listing 9-2.

Listing 9-2 The WALLDEMO.CPP program

```
// WALLDEMO.CPP
//
// Calls ray-casting function to draw mono-colored
// view of maze.
//
// Written by Christopher Lampton for
// Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "screen.h"
#include  "pcx.h"
#include  "wallcast.h"

map_type map={
 {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,1,1,0,1,0,0,0,0,1},
 {1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1},
 {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
};

float viewing_angle=0;
int viewer_height=32;
int xview=8*64;
int yview=8*64;

void main(int argc,char* argv[])
{
  // Read arguments from command line if present:
```

continued on next page

continued from previous page

```
  if (argc==2) viewing_angle=atof(argv[1]);

  // Point variable at video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Set mode 13h:
  setmode(0x13);

  // Clear the screen:
  cls(screen);

  // Draw a raycast view of the maze:
  draw_maze(map,screen,xview,yview,viewing_angle,viewer_height);

  // Wait for user to hit a key:
  while (!kbhit());

  // Reset video mode and exit:
  setmode(oldmode);
}
```

Running WALLDEMO.EXE

To use this program, go to the RAYCAST directory and type WALLDEMO. A view of a solid-colored ray-cast maze, as viewed from a position at the center of the maze by a viewer turned to angle 0, will be drawn on the display (see Figure 9-11). Press any key to return to DOS. To view the maze from other angles, type a floating point number on the command line following the program name. Remember that the angle must be expressed in radians, so values should be in the range 0 to 6.28. Figure 9-12 shows the maze from an angle of 0.5 radians. Figure 9-13 shows the maze from an angle of 3 radians.

Although the maze depicted by WALLDEMO is solid-colored, the results are still quite striking. Using a method quite different from any employed in previous chapters of this book, we've produced an image of the maze every bit as vivid as those produced by the polygon-based code in chapter 5. And yet at no point have we needed to deal with such issues as hidden-surface removal or clipping polygons against the left and right edge of the viewport. (We still have to deal with the issue of clipping against the top edge of the viewport.) In fact, hidden surface removal is taken care of automatically and elegantly by the ray-casting procedure, because the rays will simply never strike hidden surfaces. And there's no reason to clip against the left and right edges of the viewport because we never cast any rays that go beyond these edges.

Furthermore, the fine coordinate system that we are using to specify the position of the viewer is sufficiently high in resolution that we can potentially use this technique to create animations in which the movement is as smooth as that in a flight simulator, which was our avowed goal at the beginning of this book. (This version of the ray-casting code really isn't fast enough to produce satisfactory animation frame rates, but we'll develop an optimized version in chapter 12 that will run at considerably faster speeds.)

Still, the ray-cast image of the maze produced by this program isn't nearly as vivid as the bitmapped images introduced back in chapter 4. If all that we can do with ray casting is to create solid-colored images of a maze, the technique will be much too limited to create maze games with.

Fortunately, we can do a great deal more than this with ray casting. Let's add some more features to our *draw_maze()* function, starting with texture mapping.

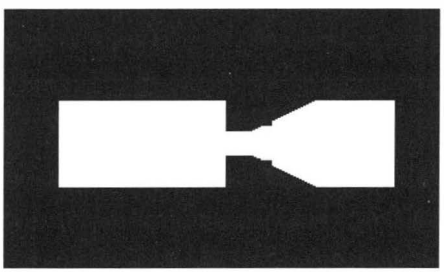


Figure 9-11 A solid-colored ray-cast view of the maze as produced by the WALLDEMO program

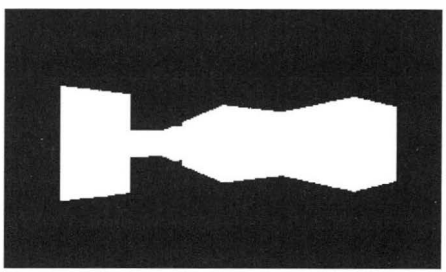


Figure 9-12 The same maze from an angle of 0.5 radians

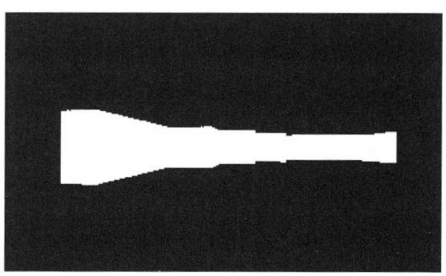


Figure 9-13 ...and from an angle of 3 radians

Ray Casting and Texture Mapping

Texture-mapping techniques, which allow us to cover the walls of the maze with scaled, bitmapped images, can easily be added to our ray-casting code. In fact, we already have almost all of the code that we'll need. Back in chapter 6, I demonstrated a technique for drawing a texture-mapped polygon as a series of vertical columns, where each column was scaled from the equivalent column of a prestored bitmap. The code that we used in that chapter to draw the individual columns of the bitmap can be adapted with minimal change to draw texture-mapped walls in a ray-casting program. The wallcasting code that we just developed tells us the height of the portion of the wall corresponding to a given column of pixels on the display. We can then use the texture-mapping code to draw the equivalent column of pixels from a predrawn bitmap, scaling it to the height of the wall column.

In order to incorporate bitmaps into our program, we need a file format in which to store the bitmaps on the disk and an internal format for storing the bitmaps in the computer's memory. For the first, we'll use PCX files, since we've already created code for reading this format. For the second, we'll use the PCX_struct format we developed back in chapter 4. This internal format is a little wasteful of memory, since it devotes additional space to PCX header information that we don't really need, but it's adequate for the demonstration that we're going to write in this chapter. The PCX file, which is in the RAYCAST directory as IMAGES.PCX, contains fifteen 64 by 64 images arranged in three rows of five images. The texture-mapping code will extract these images directly from the *image* field of the PCX structure.

The program that calls our textured-mapped *draw_maze()* function (which is in the RAYCAST directory on the disk as TEXTDEMO.CPP) will need to load the IMAGES.PCX file (or any other PCX file containing a set of 64 by 64 images arranged in this format) using the *loadPCX()* function from chapter 4:

```
// Load texture map images:
  if (loadPCX("images.pcx",&textmaps)) exit(1);
```

The calling program will also need to maintain the map of the maze, as in previous demos. However, the format will be slightly different this time. Earlier, we used zeros to indicate empty maze squares and non-zero numbers to represent maze squares that are occupied by cubes. Now we're going to use specific numbers to indicate which texture is to be mapped onto the walls of a cube. Zeros will still indicate empty squares, but now non-zero numbers will indicate which of the 15 textures in the IMAGES.PCX file we are going to use. The 15 images will be numbered, starting at 1 in the upper-left corner and continuing

left to right and top to bottom through the image, as shown in Figure 9-14. Figure 9-14 shows the map we'll be using in the demo:

```
map_type walls={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 7, 5, 0, 5, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 5},
 { 7, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 5},
 { 7, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 5},
 { 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 4, 0, 0, 0, 5},
 {12, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,11, 0, 0, 0, 5},
 {11, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,10, 0, 0, 5},
 {11, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,10, 0, 0, 5},
 {11, 7, 7, 0, 0, 0, 0, 0, 0, 0, 0,11,10, 0, 0, 5},
 {12, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,11, 0, 0, 0, 5},
 {12, 7, 7, 0, 0, 0, 5, 1, 0, 1, 0, 0, 0, 0, 0, 5},
 { 7, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 5},
 { 7, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,12},
 { 7, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,11},
 { 7, 6, 6, 6, 6, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,10},
 { 7, 5, 5, 5, 5, 5, 5,15,15,15, 5, 5, 5, 5, 5, 5}
};
```

The textured *draw_maze()* function, which is in the file TEXTCAST.CPP in the RAYCAST directory, has a slightly revised prototype in TEXTCAST.H:

```
void draw_maze(map_type map,char far *screen,int xview,
               int yview,float viewing_angle,
               int viewer_height,char far * textmaps);
```

We've now added a pointer to a *char* array called *textmaps* as the last entry in the parameter list. It should point to a 64,000-byte buffer containing the 64 by 64 images to be mapped onto the walls of the maze. The calling program will call it with a pointer to the *image* field of the PCX structure:

```
draw_maze(walls,screen,xview,yview,viewing_angle,
          viewer_height,textmaps.image);
```

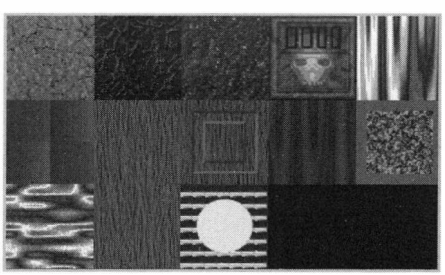


Figure 9-14 The 15 image positions in the PCX file

Finding the Bitmap Column

Much of the texture-mapped *draw_maze()* function will be identical to the solid-color one that we introduced earlier in this chapter. As before, it will be constructed as a large *for()* loop that steps through all of the vertical columns in the viewport. The process of casting the ray will once again start out by calculating the angle of the current ray, in radians, relative to a ray traveling due north at 0 radians. It will then create an arbitrary endpoint for the ray, rotate it to the current angle, and translate the coordinates of the starting and ending points of the ray relative to the viewer's position in the maze. The ray will be cast across the x and y grid lines until a wall is encountered.

The drawing of the wall will be handled a bit differently by this function. When the point of intersection between the ray and the wall is calculated, a separate value will be calculated indicating the precise horizontal point on the face of the cube where the ray strikes. This value will be measured in fine coordinates, with the far left side of the cube face being at coordinate 0 and the far right side of the cube face being at coordinate 63, as shown in Figure 9-15. This value is simply the x or y coordinate (depending on whether the face is on an x or y grid line) taken modulo 64 — that is, the remainder after the coordinate has been divided by 64. We can obtain this value quickly by ANDing the relevant coordinate with 63 (hexadecimal 3F). If the ray strikes a face aligned with an x grid line, we'll AND the y coordinate to obtain the value:

```
tmcolumn = (int)y & 0x3f;
```

If the ray strikes a face aligned with a y grid line, we'll AND the x coordinate to obtain the value:

```
tmcolumn = (int)x & 0x3f;
```

Why are we doing this? The integer variable *tmcolumn* (which stands for "texture map column") is now equal to the column of the 64 by 64 bitmap to be mapped onto that vertical column of the cube face. We can now scale this column, as we did back in chapter 6, to fit the visible height of the wall at the point of intersection.

Clipping the Column

This complicates the wall clipping a bit, because we now have to reindex into the texture map if the wall column extends past the top of the viewport. At the moment, *tmcolumn* tells us where to find the pixel to be copied from the bitmap to the top of the column on the display. If no clipping were necessary, we would simply need to calculate the offset of the upper-left corner of the bitmap (which I'll show you how to do in a moment) and add *tmcolumn* to the offset to find the

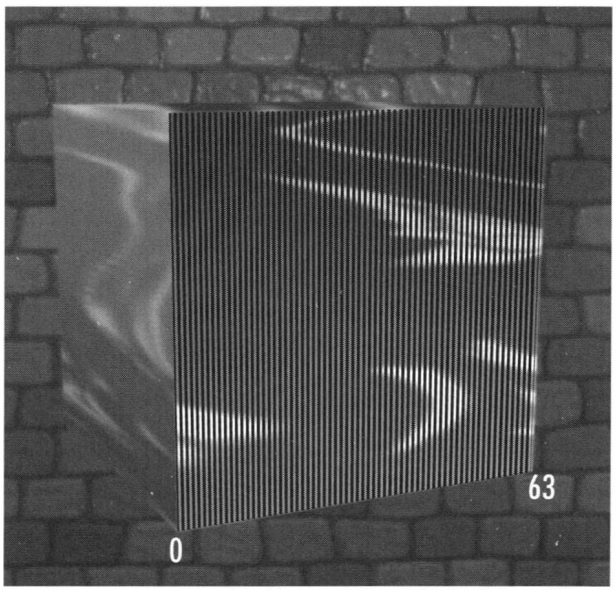


Figure 9-15 The column positions on a single cube face are numbered 0 to 63 from left to right

pixel. If the column runs off the top of the display, however, we'll need to index forward into the bitmap, as we did when clipping polygon columns in chapter 6.

First, we'll create an integer variable *t* to represent the clipped top of the column:

```
int t=tmcolumn;
```

Then we'll create a variable to hold the clipped height of the column:

```
int dheight=height;
```

Finally, we'll check to see whether the column extends past the top of the viewport and trim *dheight* if it does, just as we did in the program earlier in this chapter:

```
if (top < VIEWPORT_TOP) {
  dheight-=(VIEWPORT_TOP - top);
```

Now we need to adjust the variable *t* to point to the bitmap pixel that will be at the top of the clipped column. We've already done this in chapter 6. To reiterate, this operation is a little tricky because the pixels in the column as it will be drawn on the screen don't have a one-to-one correspondence with the pixels in the bitmap being copied to the column. If the wall is nearby, the pixels may be expanded to two or three times their normal size as the bitmap is scaled to the column size. If it's far away, some of the pixels may be skipped altogether. We

know that VIEWPORT_TOP - *top* pixels have been clipped from the top of the column on the screen, but how many pixels does this represent in the corresponding column of the bitmap? To find out, we must calculate the ratio between the visible size of the wall and the size of the bitmap:

```
float yratio=(float)height/WALL_HEIGHT;
```

Then we must multiply this value by VIEWPORT_TOP - *top* to find out how many pixels of the bitmap have been trimmed:

```
t+=(VIEWPORT_TOP-top)*yratio*320;
```

You'll notice that we multiply the resulting value by 320 in order to advance it to the proper row of the bitmap containing the image. (The image to be mapped onto the wall is only 64 pixels wide, so you might wonder why we don't multiply this by 64. The bitmap in which the image is stored, however, is 320 pixels wide, so we must advance by that amount to reach the next column of the bitmap that *t* will be pointing to.) We then advance *t* by this value.

The clipping of the bottom of the column against the bottom of the viewport proceeds exactly as it did in the last program. There's no need to clip the bitmap against the bottom, since we'll simply stop drawing when the bottom is reached, leaving the remainder of the bitmap uncopied.

The Error Term Revisited

As in chapter 6, we'll be using incremental division to determine when it is necessary to advance one more row farther into the bitmap. So we must initialize a vertical error term:

```
int tyerror=IMAGE_HEIGHT;
```

If you're hazy on how the error term works, you might want to reread chapter 6, as well as the material on Bresenham's algorithm in chapter 2. We'll see in a moment why the error term has been initialized to the height of the bitmap.

We then proceed as before, setting the integer variable *tile* equal to the value in the *wall[]* array for the square occupied by the cube that we are now drawing:

```
int tile=map[xmaze][ymaze]-1;
```

We subtract 1 from this value because there is no way that we can represent image 0 in the *wall[]* array, that number being used to represent empty squares. However, in order for the following calculations to work correctly, the images in the first row must be numbered 0 through 4, the images in the second row 5 through 9, and so forth. Subtracting 1 converts a 1 value in the map to represent image 0, a 2 to represent image 1, and so forth.

Now we need a variable that will point to the upper-left corner of the bitmap to be drawn in the *textmaps* buffer. You'll recall that the images in this buffer are arranged in three rows of five images each. To calculate which row the image is in, we can divide *tile* by 5. To get the actual offset at which this row starts, we can multiply the height of the images (in the constant IMAGE_HEIGHT, which has been set to 64) by 320 and multiply the result by the number of the row. Finally, we must take the number of the tile modulo 5 to get the position of the image in the row, multiply the result by the width of an image to get the offset within the row, and add the result to the offset of the row:

```
unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;
```

Adding *t* to the result gives us the actual offset of the next pixel to be drawn within the overall bitmap buffer.

We'll use a *for()* loop to draw the column:

```
for (int h=0; h<IMAGE_HEIGHT; h++) {
```

This loop will execute once for every pixel in the original bitmap, rather than every pixel on the display. This guarantees that we won't inadvertently overstep the bottom of the bitmap and copy pixels to the display that don't belong.

To decide if we need to draw a pixel on the next loop, we'll check to see if the value of *tyerror* is greater than or equal to the height of a column of the bitmap:

```
while (tyerror>=IMAGE_HEIGHT) {
```

Earlier, we initialized *tyerror* to 64, to the value of the IMAGE_HEIGHT constant, which guarantees that the first pixel in the column will be drawn.

If the error term is greater than or equal to IMAGE_HEIGHT, we copy the current pixel from the bitmap to the viewport:

```
screen[offset]=textmaps[tileptr];
```

We then reset the error term by subtracting IMAGE_HEIGHT from it:

```
tyerror-=IMAGE_HEIGHT;
```

and we move *offset* to point at the next line in the viewport:

```
  offset+=320;
}
```

That closes the *while()* loop. If the error term is still greater than IMAGE_HEIGHT after IMAGE_HEIGHT has been subtracted from it, the loop will execute again — and will continue executing until the error term is less than IMAGE_HEIGHT. This causes the pixels to be enlarged when the screen column is longer than the column in the bitmap.

Now we start the process all over again by adding the unclipped height of the screen column to the error term:

```
tyerror+=height;
```

If this causes the error term to become greater than the height of a bitmap column, the next pixel in the bitmap will be drawn next time the loop repeats. If not, pixels will be skipped until the cumulative adding of the *height* value to the error term pushes it over the threshold. Before we can draw the next pixel in the bitmap column, however, we must advance *tileptr* to point at it:

```
      tileptr+=320;
    }
  }
}
```

That terminates all loops. The walls have now been drawn. The complete texture-mapped ray-cast *draw_maze()* function appears in Listing 9-3.

The Texture-Mapped Ray-Cast draw_maze() Function

Listing 9-3 The draw_maze() function

```
// TEXTCAST.CPP
//
// Function to draw texture-mapped walls using ray-casting.
// Written by Christopher Lampton for
// Gardens of Imagination (Waite Group Press).
//

#include <stdio.h>
#include <math.h>
#include "textcast.h"
#include "pcx.h"
#include "distance.h"
#include "slope.h"

// Constant definitions:
const WALL_HEIGHT=64;           // Height of wall in pixels
const VIEWER_DISTANCE=192;      // Viewer distance from screen
const VIEWPORT_LEFT=0;          // Dimensions of viewport
const VIEWPORT_RIGHT=319;
const VIEWPORT_TOP=0;
const VIEWPORT_BOT=199;
const VIEWPORT_HEIGHT=VIEWPORT_BOT-VIEWPORT_TOP;
const VIEWPORT_CENTER=VIEWPORT_TOP+VIEWPORT_HEIGHT/2;
```

```
void draw_maze(map_type map,char far *screen,int xview,
                int yview,float viewing_angle,
                int viewer_height,char far * textmaps)

// Draws a ray-cast image in the viewport of the maze represented
// in array MAP[], as seen from position XVIEW, YVIEW by a
// viewer looking at angle VIEWING_ANGLE where angle 0 is due
// north. (Angles are measured in radians.)

{
  // Variable declarations:
  int sy,offset;       // Pixel y position and offset

    float column_angle=atan((float)(column-160)
                        / VIEWER_DISTANCE);
    if (column_angle==0.0) column_angle=0.001;

    // Calculate angle of ray relative to maze coordinates
    float radians=viewing_angle+column_angle;

    // Rotate endpoint of ray to viewing angle:
    int x2 = -1024 * (sin(radians));
    int y2 = 1024 * (cos(radians));

    // Translate relative to viewer's position:
    x2+=xview;
    y2+=yview;

    // Initialize ray at viewer's position:
    float x=xview;
    float y=yview;

    // Find difference in x,y coordinates along ray:
    float xdiff=x2-xview;
    float ydiff=y2-yview;

    // Cheat to avoid divide-by-zero error:
    if (xdiff==0) xdiff=0.0001;

    // Get slope of ray:
    float slope = ydiff/xdiff;

    // Cheat (again) to avoid divide-by-zero error:
    if (slope==0.0) slope=.0001;

    // Cast ray from grid line to grid line:
    for (;;) {

      // If ray direction positive in x, get next x grid line:
      if (xdiff>0) grid_x=((int)x & 0xffc0)+64;

      // If ray direction negative in x, get last x grid line:
      else grid_x=((int)x & 0xffc0) - 1;
```

continued on next page

continued from previous page

```
      // If ray direction positive in y, get next y grid line:
      if (ydiff>0) grid_y=((int)y & 0xffc0) +64;

      // If ray direction negative in y, get last y grid line:
      else grid_y=((int)y & 0xffc0) - 1;

      // Get x,y coordinates where ray crosses x grid line:
      xcross_x=grid_x;
      xcross_y=y+slope*(grid_x-x);

      // Get x,y coordinates where ray crosses y grid line:
      ycross_x=x+(grid_y-y)/slope;
      ycross_y=grid_y;

      // Get distance to x grid line:
      xd=xcross_x-x;
      yd=xcross_y-y;
      xdist=sqrt(xd*xd+yd*yd);

      // Get distance to y grid line:
      xd=ycross_x-x;
      yd=ycross_y-y;
      ydist=sqrt(xd*xd+yd*yd);

      // If x grid line is closer...
      if (xdist<ydist) {

        // Calculate maze grid coordinates of square:
        xmaze=xcross_x/64;
        ymaze=xcross_y/64;

        // Set x and y to point of ray intersection:
        x=xcross_x;
        y=xcross_y;

        // Find relevant column of texture map:
        tmcolumn = (int)y & 0x3f;

        // Is there a maze cube here? If so, stop looping:
        if (map[xmaze][ymaze]) break;
      }
      else { // If y grid line is closer:

        // Calculate maze grid coordinates of square:
        xmaze=ycross_x/64;
        ymaze=ycross_y/64;

        // Set x and y to point of ray intersection:
        x=ycross_x;
        y=ycross_y;

        // Find relevant column of texture map:
        tmcolumn = (int)x & 0x3f;
```

```
    // Is there a maze cube here? If so, stop looping:
    if (map[xmaze][ymaze]) break;
  }
}

// Get distance from viewer to intersection point:
xd=x-xview;
yd=y-yview;
distance=(long)sqrt(xd*xd+yd*yd)*cos(column_angle);
if (distance==0) distance=1;

// Calculate visible height of wall:
int height = VIEWER_DISTANCE * WALL_HEIGHT / distance;

// Calculate bottom of wall on screen:
int bot = VIEWER_DISTANCE * viewer_height
          / distance + VIEWPORT_CENTER;

// Calculate top of wall on screen:
int top = bot - height;

// Initialize temporary offset into texture map:
      int t=tmcolumn;

// If top of current vertical line is outside of
// viewport, clip it:

int dheight=height;
if (top < VIEWPORT_TOP) {
  dheight-=(VIEWPORT_TOP - top);
  float yratio=(float)height/WALL_HEIGHT;
  t+=(VIEWPORT_TOP-top)*yratio*320;
  top=VIEWPORT_TOP;
}
if (bot > VIEWPORT_BOT)
  dheight -= (bot - VIEWPORT_BOT);

// Point to video memory offset for top of line:
offset = top * 320 + column;

// Initialize vertical error term for texture map:
int tyerror=IMAGE_HEIGHT;

// Which graphics tile are we using?
int tile=map[xmaze][ymaze]-1;

// Find offset of tile and column in bitmap:
unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;

// Loop through the pixels in the current vertical
// line, advancing OFFSET to the next row of pixels
// after each pixel is drawn.
```

continued on next page

continued from previous page

```
        for (int h=0; h<IMAGE_HEIGHT; h++) {

      // Are we ready to draw a pixel?
      while (tyerror>=IMAGE_HEIGHT) {

        // If so, draw it:
        screen[offset]=textmaps[tileptr];

        // Reset error term
        tyerror-=IMAGE_HEIGHT;

        // And advance OFFSET to next screen line:
        offset+=320;
      }

      // Incremental division:
      tyerror+=height;

      // Advance TILEPTR to next line of bitmap:
      tileptr+=320;
    }
  }
}
```

To see this code in action, go to the RAYCAST directory and type TEXTDEMO. You'll see a texture-mapped view of the maze walls as they look from the center of the room at an angle of 0 radians, as shown in Figure 9-16. To see the view from other angles, type a number of radians from 0 to 6.2 after the name of the program. Figures 9-17a through e show the maze from angles of 1, 2, 3, 4, and 5 radians. You can also use fractional radian values to get more precise views of the maze.

The improvement over the earlier solid-color ray-casting program is striking. The texture-mapped walls look quite vivid and dramatic. We're well on our way to creating a topnotch ray-casting engine.

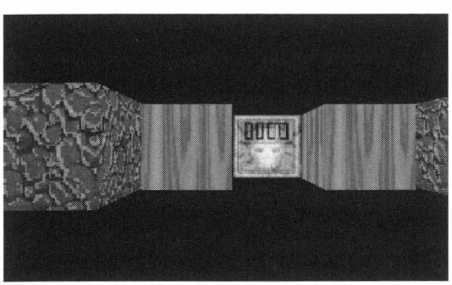


Figure 9-16 The maze viewed from an angle of 0 radians, as depicted by the TEXTDEMO program

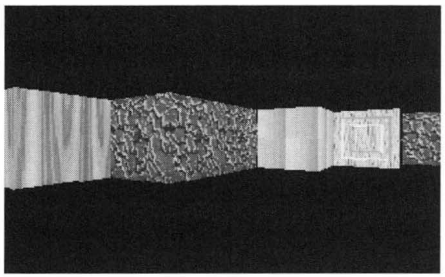


Figure 9-17a–e The maze as seen from angles of 1, 2, 3, 4, and 5 radians, respectively

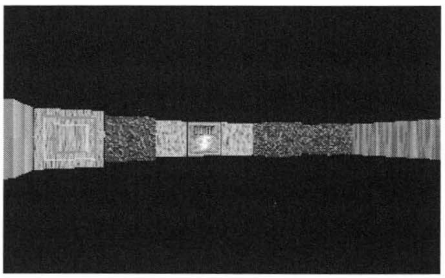


Figure 9-17b

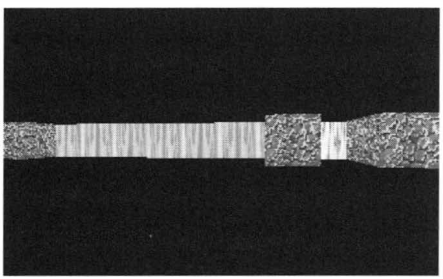


Figure 9-17c

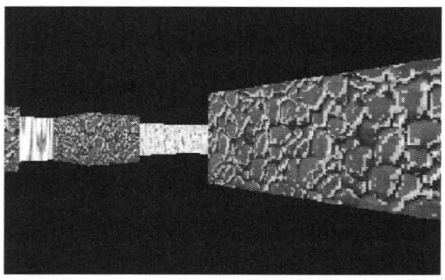


Figure 9-17d

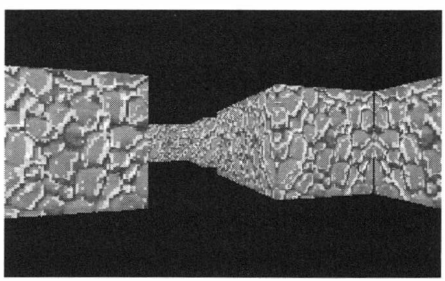


Figure 9-17e

Floorcasting

Truth to tell, we now have almost enough ray-casting code to write an arcade game along the lines of Wolfenstein 3D. All we need is some additional code for drawing bitmapped objects, such as treasure chests and Nazi guards, and we'd have a clone of the Wolf 3D engine. (Actually, this isn't quite true. The Wolfenstein 3D

engine appears to use texture-mapped polygons to represent doors and sliding walls, but we'll ignore that detail for the moment.) We can now draw the walls of a texture-mapped maze, which is pretty much all that Wolfenstein 3D does to generate scenery. (The floors and ceilings are drawn in solid colors, but these can be laid down before the ray casting begins.)

However, the technology of ray casting has proceeded apace since Wolfenstein 3D came on the scene. It's no longer enough just to display texture-mapped walls. A ray-cast game needs to display texture-mapped floors and ceilings as well. So we'll now develop a floorcasting version of our engine. The first version will drawn *only* texture-mapped floors and ceilings, then we'll create a second version that will combine the floors with the walls. Ready? Let's go.

The program that calls our floorcasting *drawmaze()* function will need to maintain *two* arrays, one to represent the map of the floor, the other to represent the map of the ceiling. Each element in these arrays will represent the number of the bitmap in the IMAGES.PCX file to be mapped onto the corresponding segment of floor or ceiling. Since each square of the maze measures 64 by 64 in the fine coordinate system, there will be a perfect correspondence between the pixels in the bitmap and the points in the fine coordinate system, with one pixel per fine coordinate point. Here are the two arrays that we'll be using:

```
map_type flor={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 8, 8, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 8, 8, 8, 8, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 8, 8, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
};

map_type ceiling={
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
```

```
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13}
};
```

We've called the first array *flor[]* instead of *floor[]* so that the linker won't mistake it for a reference to the C++ *floor()* function when we attempt to pass a pointer to it in a parameter list.

Once again, the prototype of the function, in FLOORCST.H, will need to be modified slightly from earlier versions:

```
void draw_maze(map_type floor,map_type ceiling,char far *screen,
                       int xview,int yview,
                       float viewing_angle,int viewer_height,
                       char far *textmaps);
```

We've now added the *floor* and *ceiling* arrays to the parameter list and have dropped the *wall* array, since we have no need for it in this function.

As before, the function is constructed as a large *for()* loop that iterates through all of the columns in the viewport. And we start out by calculating the angle of the ray cast through the current column relative to north, storing the resulting angle in the floating point variable *radians*. At this point, however, the floorcasting function takes off in a completely different direction.

Here's what it will do: We will start at the bottommost pixel in the column, the one at the very bottom of the viewport, and cast a ray through that column. We'll determine where it strikes the floor, in terms of the fine coordinate system, then determine the color of the pixel at that point. Then we'll draw that pixel in the viewport. We'll continue doing this for all of the pixels in the bottom half of the viewport. (Actually, we'll stop 5 pixels short of the center of the viewport, to give the impression that the more distant floor pixels have faded into darkness. When we combine the floorcasting code with the wallcasting code, we'll simply stop casting when we reach a wall.)

We'll use a second *for()* loop to step through all of the pixels in the lower portion of the column, from the one at the bottom of the viewport to the one 5 pixels short of the vertical center:

```
for (int row=VIEWPORT_BOT; row>VIEWPORT_CENTER+5; --row) {
```

The Similar Triangles Method

Now we need to determine what color the pixel in that position should be, which means that we must cast a ray through that pixel to determine where it hits the floor. But how do we do this? Look at Figure 9-18. It shows the viewer looking through the video display at the floor of the maze. The line coming out of the viewer's eyes and striking the floor is labeled B. Another line runs from the viewer's feet to the position where the ray strikes the floor. We've labeled this A. Finally, the line that runs from the viewer's eyes to the floor is labeled C.

You'll notice that these lines, taken together, form a triangle. We know the length of one side of this triangle, side C (the one that runs from the viewer's eye to the floor). (It's contained in the parameter *viewer_height* that was passed from the calling program.) We *want* to know the length of A, from which we can calculate the coordinates of the floor pixel struck by the ray. We could use trigonometry to calculate this length, but there's an easier way, one that lends itself more to optimization. It's called the *similar triangles method.*

Look at Figure 9-19. The same triangle from Figure 9-18 is depicted here, but now a second triangle has been nested inside of it. One edge of this triangle, running parallel to line C in the big triangle, is labeled *c.* This edge represents the height of the viewer's eye above the row through which we are casting. The line labeled *a* in the small triangle is the distance from the viewer to the display. It runs parallel to line A in the big triangle. And the line labeled *b* in the small triangle is the straight line distance from the viewer's eye to the row through which the ray has been cast. This line runs parallel to line B in the big triangle.

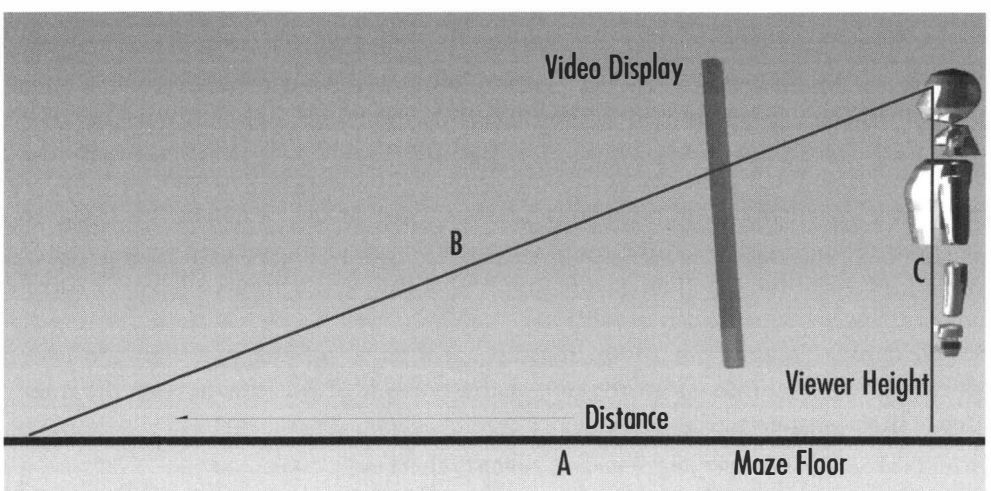


Figure 9-18 The viewer looking through the video display at the maze

Because all of the lines in the small triangle are parallel to lines in the big triangle, we know that all of the angles between sides in the small triangle must be the same as the angles between sides in the big triangle, even though the lengths of the sides are different. In mathematical terms, two triangles in which all of the angles between the sides are the same but the lengths of the sides are different are called *similar triangles*. According to the Law of Similar Triangles, the ratio between the sides of a similar triangle are always the same no matter what the actual lengths of the sides are. If one side is 1.5 times the length of another in a small triangle, it will be 1.5 times the size of the other in a larger version with the same angles. In algebraic terms, we would write:

```
A/C = a/c
```

where the letters refer to the sides of our similar triangles.

We already know the lengths of three of these sides. C is the value of the parameter *viewer_height*, *a* is the value of the constant VIEWER_DISTANCE, and *c* can be easily calculated by subtracting VIEWPORT_CENTER (the row of the viewport with which the viewer's eyes are supposedly aligned) from *row* (the row we are casting the ray through). We need to solve this equation for A. Following the standard algebraic rules, we can rewrite the equation as:

```
A = a/c * C
```

Plugging in the values from our program, we get:

```
distance_to_pixel = VIEWER_DISTANCE/(VIEWPORT_CENTER - row) *
viewer_height
```

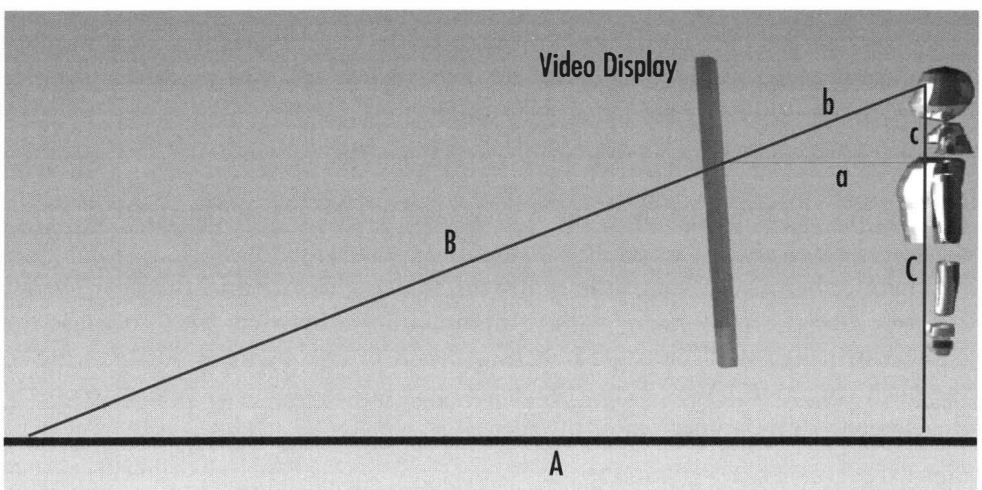


Figure 9-19 The small triangle with sides a, b, c is mathematically similar to the large triangle with sides A, B, C

In the actual program, we'll code that slightly differently. First we'll perform the division and store the value in the floating point variable *ratio,* since the result of the division is the ratio between the sides:

```
float ratio=(float)viewer_height/(row-VIEWPORT_CENTER);
```

Then we'll multiply the result by VIEWER_DISTANCE to get the actual distance of the pixel from the viewer. We'll also perform the cosine adjustment, as explained earlier in this chapter, to avoid a fish-eye effect:

```
distance=ratio*VIEWER_DISTANCE/cos(column_angle);
```

Now we'll rotate this distance around the origin to the angle that we stored earlier in the variable *radians:*

```
int x = - distance * (sin(radians));
int y = distance * (cos(radians));
```

(This is the same calculation that we used during wallcasting, explained earlier in this chapter.) Then we'll translate the resulting coordinates relative to the viewer's position:

```
x+=xview;
y+=yview;
```

In order to find the pixel color for this position in the maze, we need to know which maze grid square it's part of:

```
int xmaze = x / 64;
int ymaze = y / 64;
```

We also need to know the column of texture map that should be applied to this pixel. We'll find it by taking both the x and y coordinates positions modulo 64, and using the result to calculate the necessary index into the bitmap buffer relative to the upper-left corner of the bitmap:

```
int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);
```

Once again, we've used 320 as the bitmap width rather than IMAGE_WIDTH because the bitmaps are stored in a buffer 320 pixels wide.

The number of the image tile that we'll be using is stored in the *floor[]* array. (We've called the array *floor[]* within the function rather than *flor[],* because the context in which we'll be using it here makes it clear that we're not referring to the *floor()* function.) We can use the *xmaze* and *ymaze* values, as before, to obtain the element with the number of the tile:

```
int tile=floor[xmaze][ymaze];
```

We'll calculate the offset of the upper-right corner of the bitmap exactly as before:

```
unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;
```

Similarly, we'll calculate the video memory offset of the current row in the current column using the standard calculation:

```
offset=row*320+column;
```

Finally, we'll copy the pixel from the bitmap to the viewport:

```
  screen[offset]=textmaps[tileptr];
}
```

And that's all there is to drawing the floor.

Drawing the Ceiling

We'll draw the ceiling almost exactly as we drew the floor. The easiest way to draw the ceiling would be to draw it at the same time as we draw the floor, mirroring every floor pixel with a ceiling pixel in the upper half of the viewport. That would be faster than what we're actually going to do, but it would assume that the viewer's eye height is fixed midway between the floor and ceiling. This will often be the case — in fact, many ray casters work on this assumption — but we'll make the ray caster that we develop in this chapter a bit more versatile. That's why we've passed *viewer_height* as a parameter from the calling program, so that it can be varied dynamically if necessary. But this means we'll have to cast the ceiling separately, recalculating all of the numbers from scratch. Instead of starting with the bottommost pixel in the viewport and advancing until we're 5 pixels below the center, we'll start with the topmost pixel in the viewport and advance until we're 5 pixels above the center:

```
for (row=VIEWPORT_TOP; row<VIEWPORT_CENTER-5; row++) {
```

We can use the similar triangles method to calculate the distance to the ceiling pixel just as we did when we calculated the distance to the floor pixel, except that the triangles will be upside down. Instead of dividing *viewer_height* from VIEWPORT_CENTER-*row* to get the ratio of the two sides, we'll divide WALL_HEIGHT-*viewer_height* (the distance from the viewer's eyes to the maze ceiling) by VIEWPORT_CENTER-*row* (the distance from the viewer's eyes to the row through which we're casting the ray):

```
float ratio=(float)(WALL_HEIGHT-viewer_height)
              / (VIEWPORT_CENTER-row);
```

We can then calculate the distance to the floor pixel and the coordinates of the floor pixel just as we did for the floor. However, we'll need to take the tile number from the *ceiling[]* array rather than the *floor[]* array:

```
int tile=ceiling[xmaze][ymaze];
```

The remaining code is identical to the code for casting the floor. And that's the rest of the floorcasting *drawmaze()* function. The complete text of the function is in Listing 9-4.

The Floorcasting draw_maze() Function

Listing 9-4 The floorcasting draw_maze() function

```
//
// FLOORCST.CPP
// Draws texture mapped floor
// Written by Christopher Lampton
// for Gardens of Imagination (Waite Group Press)

#include <stdio.h>
#include <math.h>
#include "floorcst.h"
#include "pcx.h"

const WALL_HEIGHT=64;           // Height of wall in pixels
const VIEWER_DISTANCE=128;      // Viewer distance from screen
const VIEWPORT_LEFT=0;          // Dimensions of viewport
const VIEWPORT_RIGHT=319;
const VIEWPORT_TOP=0;
const VIEWPORT_BOT=199;
const VIEWPORT_HEIGHT=VIEWPORT_BOT-VIEWPORT_TOP;
const VIEWPORT_CENTER=VIEWPORT_TOP+VIEWPORT_HEIGHT/2;
const ZOOM=2;


void draw_maze(map_type floor,map_type ceiling,char far *screen,
                       int xview,int yview,
                       float viewing_angle,int viewer_height,
                       char far *textmaps)
{
  int y_unit,x_unit; // Variables for amount of change
                     //  in x and y
  int distance,old_distance,xd,yd,sx,sy,offset;

  // Loop through all columns of pixels in viewport:
  for (int column=VIEWPORT_LEFT; column<VIEWPORT_RIGHT; column++) {
```

```
// Calculate horizontal angle of ray relative to
//  center ray:
float column_angle=atan((float)((column-160)/ZOOM)
                    / VIEWER_DISTANCE);

// Calculate angle of ray relative to maze coordinates
float radians=viewing_angle+column_angle;

// Step through floor and ceiling pixels:
for (int row=VIEWPORT_BOT; row>VIEWPORT_CENTER+5; --row) {

  // Get ratio of viewer's height to pixel height:
  float ratio=(float)viewer_height/(row-VIEWPORT_CENTER);

  // Get distance to visible pixel:
  distance=ratio*VIEWER_DISTANCE/cos(column_angle);

  // Rotate distance to ray angle:
  int x = - distance * (sin(radians));
  int y = distance * (cos(radians));

  // Translate relative to viewer coordinates:
  x+=xview;
  y+=yview;

  // Get maze square intersected by ray:
  int xmaze = x / 64;
  int ymaze = y / 64;

  // Find relevant column of texture map:
  int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);

  // Which graphics tile are we using?
  int tile=floor[xmaze][ymaze];

  // Find offset of tile and column in bitmap:
  unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
    *IMAGE_WIDTH+t;

  // Calculate video offset of floor pixel:
  offset=row*320+column;

  // Draw pixel:
  screen[offset]=textmaps[tileptr];
}

// Step through ceiling pixels:
for (row=VIEWPORT_TOP; row<VIEWPORT_CENTER-5; row++) {

  // Get ratio of viewer's height to pixel height:
  float ratio=(float)(WALL_HEIGHT-viewer_height)
                / (VIEWPORT_CENTER-row);
```

continued on next page

continued from previous page

```
      // Get distance to visible pixel:
      distance=ratio*VIEWER_DISTANCE/cos(column_angle);

      // Rotate distance to ray angle:
      int x = - distance * (sin(radians));
      int y = distance * (cos(radians));

      // Translate relative to viewer coordinates:
      x+=xview;
      y+=yview;

      // Get maze square intersected by ray:
      int xmaze = x / 64;
      int ymaze = y / 64;

      // Find relevant column of texture map:
      int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);

      // Which graphics tile are we using?
      int tile=ceiling[xmaze][ymaze];

      // Find offset of tile and column in bitmap:
      unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
        *IMAGE_WIDTH+t;

      // Calculate video offset of floor pixel:
      offset=row*320+column;

      // Draw pixel:
      screen[offset]=textmaps[tileptr];
    }
  }
}
```

To see what this function does, go to the RAYCAST directory and type FLOORDEM. You'll see a view of a texture-mapped maze floor and ceiling stretching out into the distance (see Figure 9-20). I've deliberately used the same tile repeatedly in both the floor and the ceiling because this is how floors normally look. If you turn to an angle of approximately 3 radians — by typing the number 3 after the name of the program when executing it — you'll see a cross on the floor constructed out of an alternate bitmap (see Figure 9-21). At some angles, you may notice some oddly colored floor tiles off in the distance. These are drawn when the floorcasting algorithm works its way off the ends of the *floor[]* and *ceiling[]* arrays, since we've performed no bounds checking to keep this from happening. When we add the walls, it won't be necessary to check for boundary violations, since the floorcasting will stop when a wall is encountered.

So let's add some walls, by combining our floorcasting code with our texture-mapped wallcasting code.

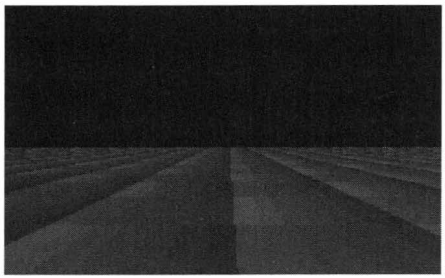


Figure 9-20 A texture-mapped ray-cast floor and ceiling as depicted by the FLOORDEM program

Figure 9-21 A cross of tiles in the middle of the floor

A Complete Ray-Casting Engine

Combining the wallcasting and floorcasting code is a relatively trivial task. We'll need to modify the prototype for the *draw_maze()* function yet again, so that we can pass pointers to *three* two-dimensional arrays: the *wall[]* array, the *floor[]* array, and the *ceiling[]* array:

```
void draw_maze(map_type map,map_type floor,map_type ceiling,
               char far *screen,int xview,int yview,
               float viewing_angle,int viewer_height,
               char far * textmaps)
```

In the demo program that we create (which is in the module RAYDEMO.CPP), we'll use the same arrays for the walls, floors, and ceilings that we've used in our earlier demos. The *draw_maze()* function itself will be in the module RAYCAST.CPP and the prototype in RAYCAST.H.

In writing this new version of the *draw_maze()* function, the major decision we find ourselves faced with is the order in which to perform the operations. Should we cast for the walls first or draw the floors first?

This isn't a difficult decision to make. If we draw the floors first, we'll need to draw the walls over them, which means that a lot of the screen pixels will be drawn twice. That's time-consuming. Even though the code that we're writing in this chapter isn't optimized for speed, we want to construct the function in such a way that it can be easily optimized later. So we'll draw the walls first, then draw the ceilings and floors only in the areas where no walls have been drawn.

We'll start out by casting the walls exactly as we did in the WALLCAST.CPP module earlier in this chapter. When we finish drawing the walls, the row number of the bottommost pixel in the wall column will be stored in the integer variable *bot*, and the row number of the topmost pixel will be stored in the integer variable *top*. Instead of drawing the floor pixels from the bottom of

the viewport to the center, we'll start drawing just below *bot* and work down to the bottom of the viewport:

```
for (int row=bot+1; row<=VIEWPORT_BOT; row++) {
```

And when we draw the ceiling, we'll start at the pixel just above *top* and work our way to the top of the viewport:

```
for (row=top-1; row>=VIEWPORT_TOP; --row) {
```

That's just about the only part of the code that needs to be altered. The resulting floorcasting, wallcasting version of the *draw_maze()* function is in Listing 9-5.

The Floorcasting, Wallcasting draw_maze() Function

Listing 9-5 The floorcasting, wallcasting draw_maze() function

```
// RAYCAST.CPP
//
// Function to draw texture-mapped walls, floors and
// ceilings using ray casting.
// Written by Christopher Lampton for
// Gardens of Imagination (Waite Group Press).
//

#include <stdio.h>
#include <math.h>
#include "raycast.h"
#include "pcx.h"
#include "distance.h"
#include "slope.h"

// Constant definitions:
const WALL_HEIGHT=64;           // Height of wall in pixels
const VIEWER_DISTANCE=192;      // Viewer distance from screen
const VIEWPORT_LEFT=0;          // Dimensions of viewport
const VIEWPORT_RIGHT=319;
const VIEWPORT_TOP=0;
const VIEWPORT_BOT=199;
const VIEWPORT_HEIGHT=VIEWPORT_BOT-VIEWPORT_TOP;
const VIEWPORT_CENTER=VIEWPORT_TOP+VIEWPORT_HEIGHT/2;

void draw_maze(map_type map,map_type floor,map_type ceiling,
                char far *screen,int xview,int yview,
                float viewing_angle,int viewer_height,
                char far * textmaps)
```

```
// Draws a ray cast image in the viewport of the maze represented
// in array MAP[], as seen from position XVIEW, YVIEW by a
// viewer looking at angle VIEWING_ANGLE where angle 0 is due
// north. (Angles are measured in radians.)

{
  // Variable declarations:
  int sy,offset;       // Pixel y position and offset
  float xd,yd;         // Distance to next wall in x and y
  int grid_x,grid_y;   // Coordinates of x and y grid lines
  float xcross_x,xcross_y; // Ray intersection coordinates
  float ycross_x,ycross_y;
  unsigned int xdist,ydist; // Distance to x and y grid lines
  int xmaze,ymaze;     // Map location of ray collision
  int distance;  // Distance to wall along ray
  int tmcolumn;        // Column in texture map
  float yratio;

  // Loop through all columns of pixels in viewport:
  for (int column=VIEWPORT_LEFT; column<VIEWPORT_RIGHT; column++) {

    // Calculate horizontal angle of ray relative to
    //  center ray:
    float column_angle=atan((float)(column-160)
                        / VIEWER_DISTANCE);

    // Calculate angle of ray relative to maze coordinates
    float radians=viewing_angle+column_angle;

    // Rotate endpoint of ray to viewing angle:
    int x2 = -1024 * (sin(radians));
    int y2 = 1024 * (cos(radians));

    // Translate relative to viewer's position:
    x2+=xview;
    y2+=yview;

    // Initialize ray at viewer's position:
    float x=xview;
    float y=yview;

    // Find difference in x,y coordinates along ray
    int xdiff=x2-xview;
    int ydiff=y2-yview;

    // Cheat to avoid divide-by-zero error:
    if (xdiff==0) xdiff=1;

    // Get slope of ray:
    float slope = (float)ydiff/xdiff;

    // Cheat (again) to avoid divide-by-zero error:
    if (slope==0.0) slope=.0001;
```

continued on next page

continued from previous page

```
// Cast ray from grid line to grid line:
for (;;) {

  // If ray direction positive in x, get next x grid line:
  if (xdiff>0) grid_x=((int)x & 0xffc0)+64;

  // If ray direction negative in x, get last x grid line:
  else grid_x=((int)x & 0xffc0) - 1;

  // If ray direction positive in y, get next y grid line:
  if (ydiff>0) grid_y=((int)y & 0xffc0) +64;

  // If ray direction negative in y, get last y grid line:
  else grid_y=((int)y & 0xffc0) - 1;

  // Get x,y coordinates where ray crosses x grid line:
  xcross_x=grid_x;
  xcross_y=y+slope*(grid_x-x);

  // Get x,y coordinates where ray crosses y grid line:
  ycross_x=x+(grid_y-y)/slope;
  ycross_y=grid_y;

  // Get distance to x grid line:
  xd=xcross_x-x;
  yd=xcross_y-y;
  xdist=sqrt(xd*xd+yd*yd);

  // Get distance to y grid line:
  xd=ycross_x-x;
  yd=ycross_y-y;
  ydist=sqrt(xd*xd+yd*yd);

  // If x grid line is closer...
  if (xdist<ydist) {

    // Calculate maze grid coordinates of square:
    xmaze=xcross_x/64;
    ymaze=xcross_y/64;

    // Set x and y to point of ray intersection:
    x=xcross_x;
    y=xcross_y;

    // Find relevant column of texture map:
    tmcolumn = (int)y & 0x3f;

    // Is there a maze cube here? If so, stop looping:
    if (map[xmaze][ymaze]) break;
  }
```

```
  else { // If y grid line is closer:

    // Calculate maze grid coordinates of square:
    xmaze=ycross_x/64;
    ymaze=ycross_y/64;

    // Set x and y to point of ray intersection:
    x=ycross_x;
    y=ycross_y;

    // Find relevant column of texture map:
    tmcolumn = (int)x & 0x3f;

    // Is there a maze cube here? If so, stop looping:
    if (map[xmaze][ymaze]) break;
  }
}

// Get distance from viewer to intersection point:
xd=x-xview;
yd=y-yview;
distance=(long)sqrt(xd*xd+yd*yd)*cos(column_angle);
if (distance==0) distance=1;

// Calculate visible height of wall:
int height = VIEWER_DISTANCE * WALL_HEIGHT / distance;

// Calculate bottom of wall on screen:
int bot = VIEWER_DISTANCE * viewer_height
          / distance + VIEWPORT_CENTER;

// Calculate top of wall on screen:
int top = bot - height;

// Initialize temporary offset into texture map:
      int t=tmcolumn;

// If top of current vertical line is outside of
// viewport, clip it:
int dheight=height;
int iheight=IMAGE_HEIGHT;
yratio=(float)WALL_HEIGHT/height;
if (top < VIEWPORT_TOP) {
  dheight-=(VIEWPORT_TOP - top);
  t+=(int)((VIEWPORT_TOP-top)*yratio)*320;
  iheight -= ((VIEWPORT_TOP-top)*yratio);
  top=VIEWPORT_TOP;
}
if (bot > VIEWPORT_BOT) {
  dheight -= (bot - VIEWPORT_BOT);
  iheight -= (bot - VIEWPORT_BOT)*yratio;
```

continued on next page

continued from previous page

```
    bot=VIEWPORT_BOT;
  }

  // Point to video memory offset for top of line:
  offset = top * 320 + column;

  // Initialize vertical error term for texture map:
  int tyerror=64;

  // Which graphics tile are we using?
  int tile=map[xmaze][ymaze]-1;

  // Find offset of tile and column in bitmap:
  unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;

  // Loop through the pixels in the current vertical
  // line, advancing OFFSET to the next row of pixels
  // after each pixel is drawn.
  for (int h=0; h<iheight; h++) {

    // Are we ready to draw a pixel?
       while (tyerror>=IMAGE_HEIGHT) {

      // If so, draw it:
      screen[offset]=textmaps[tileptr];

      // Reset error term:
      tyerror-=IMAGE_HEIGHT;

      // And advance OFFSET to next screen line:
      offset+=320;
    }

    // Incremental division:
       tyerror+=height;

    // Advance TILEPTR to next line of bitmap:
    tileptr+=320;
  }

  // Step through floor pixels
  for (int row=bot+1; row<=VIEWPORT_BOT; row++) {

    // Get ratio of viewer's height to pixel height:
    float ratio=(float)viewer_height/(row-100);

    // Get distance to visible pixel:
    distance=ratio*VIEWER_DISTANCE/cos(column_angle);

    // Rotate distance to ray angle:
```

```
  int x = - distance * (sin(radians));
  int y = distance * (cos(radians));

  // Translate relative to viewer coordinates:
  x+=xview;
  y+=yview;

  // Get maze square intersected by ray:
  int xmaze = x / 64;
  int ymaze = y / 64;

  // Find relevant column of texture map:
  int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);

  // Which graphics tile are we using?
  int tile=floor[xmaze][ymaze];

  // Find offset of tile and column in bitmap:
  unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;

  // Calculate video offset of floor pixel:
  offset=row*320+column;

  // Draw pixel:
  screen[offset]=textmaps[tileptr];
}

// Step through ceiling pixels:
for (row=top-1; row>=VIEWPORT_TOP; --row) {

  // Get ratio of viewer's height to pixel height:
  float ratio=(float)(WALL_HEIGHT-viewer_height)/(100-row);

  // Get distance to visible pixel:
  distance=ratio*VIEWER_DISTANCE/cos(column_angle);

  // Rotate distance to ray angle:
  int x = - distance * (sin(radians));
  int y = distance * (cos(radians));

  // Translate relative to viewer coordinates:
  x+=xview;
  y+=yview;

  // Get maze square intersected by ray:
  int xmaze = x / 64;
  int ymaze = y / 64;

  // Find relevant column of texture map:
  int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);
```

continued on next page

continued from previous page

```
      // Which graphics tile are we using?
      int tile=ceiling[xmaze][ymaze];

      // Find offset of tile and column in bitmap:
      unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
      *IMAGE_WIDTH+t;

      // Calculate video offset of floor pixel:
      offset=row*320+column;

      // Draw pixel:
      screen[offset]=textmaps[tileptr];
    }
  }
}
```

To run the program, go to the RAYCAST directory and type RAYDEMO. You'll see a fully texture-mapped view of the maze from an angle of 0 radians (see Figure 9-22). To see how the maze looks from other angles, type a number of radians from 0 to 6.2 after the name of the program. Figures 9-23a through e show the maze from angles of 1, 2, 3, 4, and 5 radians.

The texture-mapped maze drawn by this program is amazingly realistic, considering that it started life as little more than a trio of two-dimensional arrays and a PCX file containing 15 bitmaps. Such is the power of ray casting. Though the technique itself is fairly simple, it can be used to create vivid images very rapidly, as we'll see when we optimize our ray-casting code in chapter 13.

We now have the core of a ray-casting engine that could be used to produce a ray-cast game such as Origin's Shadowcaster or Apogee's Blake Stone and the Aliens of Gold. We can draw walls, floors, and ceilings, which is all the scenery generating that many state-of-the-art ray-casting games can do. In the next chapter, though, we're going to look at an alternate method that can produce even more realistic images than this. This method will place us on the cutting edge of ray-casting technology and will put us in a position to write a ray-casting engine that will rival the most advanced on the market.

This new method is called *heightmapping*.

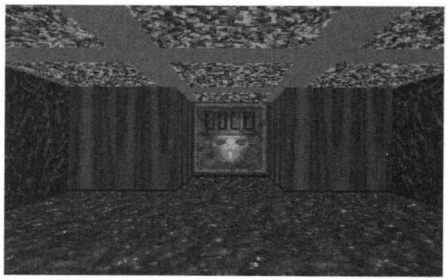


Figure 9-22 A fully texture-mapped maze viewed from an angle of 0 radians as depicted by the RAYDEMO program

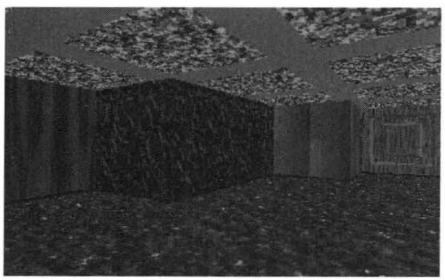


Figure 9-23a–e The maze as viewed from angles of 1, 2, 3, 4, and 5 radians, respectively

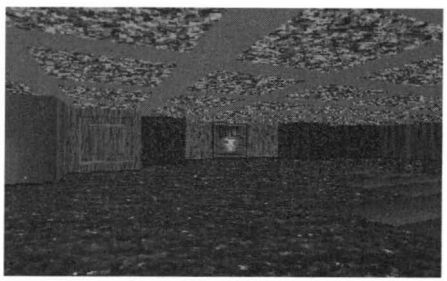


Figure 9-23b

Figure 9-23c

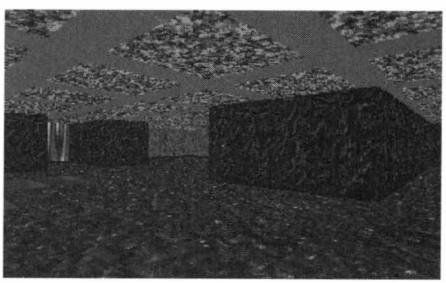


Figure 9-23d

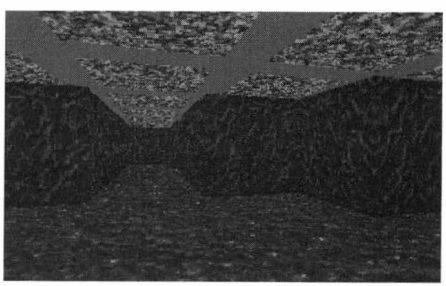
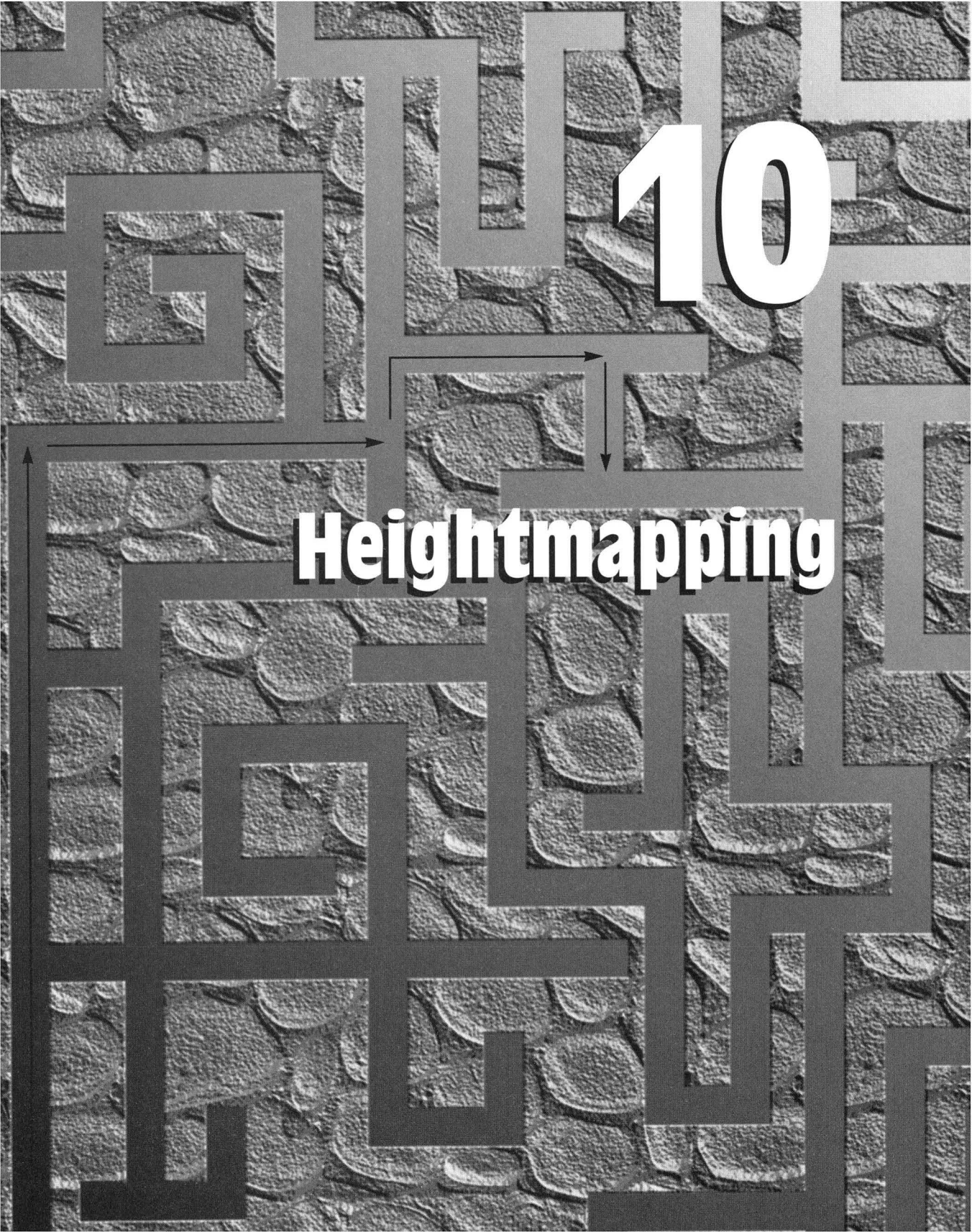


Figure 9-23e



10

Heightmapping

10

The ray-casting method described in the last chapter produces vivid images, and can do so rapidly enough for high-speed animation if properly optimized. But it has a major drawback. It ties us into a single type of scenery, a maze of texture-mapped blocks with texture-mapped floors and ceilings. Of course, that's the same type of scenery that we've been using throughout this book — indeed, that type of maze is at the very heart of the maze game genre. But even the most devoted maze game aficionado will eventually get sick of wandering down hallways that look like they were laid out on a sheet of graph paper. Before that happens, we'd better learn to inject some variety into the design of our maze game graphics.

Are there any ray-cast maze games on the market that defy this convention and feature mazes that don't look like they were drawn with a straight-edge and T square? Yes. At the time this book was written, the most notable of this elite company of games was Doom (see Figure 10-1), published by Id Software, the same company that introduced ray casting to the arcade game genre with Wolfenstein 3D. Doom features dazzlingly diverse maze graphics in which walls shoot out at all angles and have variable thicknesses. Not only do the walls in Doom defy the conventions of ray-cast maze games, but so do the floors, some of which rise to precipitous heights above the "ground." In fact, unlike any ray-cast games published before it, Doom actually lets the player walk on top of the walls!

Figure 10-1 A screen shot from Doom, published by Id Software

How does Doom do it? Does it use an advanced, hitherto unheard of form of graphics generation? That's a difficult question to answer, since I don't know the secrets of the Doom programmers. All I know about Doom is what I can see on the screen of the computer when I play the game. However, I strongly suspect that Doom is using a technique known as heightmapping.

Block-Aligned Heightmaps

The simplest form of heightmapping uses so-called *block-aligned heightmaps*. This isn't the method used by Doom, which achieves graphic effects that would not be possible if the heightmapping were strictly block aligned. However, the very simplicity of this method makes it an excellent introduction to heightmapping, so we'll talk about it first, before we move on to more advanced techniques.

Up until now, we've assumed that the walls of the cubes that make up our mazes stretched from floor to ceiling. We couldn't see the tops of these walls because they were above our field of vision (and even if we set the viewer's eye height to a value greater than the wall height, there would have been no tops to see, since our maze-drawing code contained no provisions for drawing these tops). In this chapter we'll drop that assumption. We'll allow walls to be any height at all, up to a preset limit. We'll set this limit at 256 "height units," since that will allow us to store the height of each wall section in a single byte of computer memory. And if the top of a wall is below the viewer's eye level, we'll draw the top, using the same techniques that we used in the last chapter to draw floors.

Block-aligned heightfields are much like the maze cubes that we used in the ray-casting engine in the last chapter, in that they are represented in the

computer as a two-dimensional grid of "maze squares." The difference is that each square has an associated height value. While our earlier maze programs have treated the floor of the maze as though it were flat, in a heightmapped maze each section of floor can have a different height from the sections around it. And a maze square that is higher than the the squares surrounding it will have "sides," which form the walls of the heightmapped maze. Because the height of the floor squares is variable, so is the height of the walls. Walls can be short enough to trip over or too tall to climb. In a game based on such a heightmapped ray-casting engine, short walls can be used to represent stair steps, tables, or fences. Tall walls can represent barriers or towering obelisks.

The floorheight[] Array

We'll store the height values of the maze squares in a 16 by 16 array of unsigned integer values, just like the ones we used in the last chapter to store bitmap values for the floor, ceiling, and walls of the maze. We'll still maintain the floor and wall arrays, just as we did before (though for now we'll drop the ceiling array, to simplify the code). But this time we'll add an array called *floorheight[]* to store the height values. Here's the *floorheight[]* array for the simple heightmapping demo that we'll develop over the next few pages:

```
map_type floorheight={
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20, 0, 0,20,20,20,20,20,20,20,40,60,80,100,120,120},
 {20, 0, 0,20,20,20,20,20,20,20,20,20,20,20,120,120},
 {20, 0, 0,20,20,20,20,20,20,20,20,20,20,20,120,120},
 {20,20,20,20,20,20,20,20,20,20,40,60,80,100,120,120},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20,20,20,20,60,60,60,60,60,20,20,20,20,20,20,20},
 {20,20,20,20,60,100,100,100,60,20,20,20,20,40,20,20},
 {20,20,20,20,60,100,150,100,60,20,20,20,40,40,20,20},
 {20,20,20,20,60,100,100,100,60,20,20,40,40,40,20,20},
 {20,20,20,20,60,60,60,60,60,20,20,20,20,20,20,20},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20}
};
```

You'll notice that most of the "floor squares" in this map have been set at a height of 20 units. By making 20 units the "normal" floor height throughout the maze, we can create depressions in the floor as well as walls. For instance, you may have noticed that some of the floor squares in the middle left portion of the array have a value of 0. These lowered floor segments, arranged in a rectangular relationship,

will be used to create a swimming pool in the maze. A swimming pool? Be patient. You'll get a look at this unusual maze convenience in a moment.

The wall[] and flor[] Arrays

Here are the *wall[]* and *flor[]* arrays that we'll be using:

```
map_type wall={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5,14,14, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 {14, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 {14, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 {14, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5,14,14, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,12,12,12,12,12, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,12,12,12,12,12, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,12,12,12,12,12, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,12,12,12,12,12, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,12,12,12,12,12, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
};

map_type flor={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5,10,10, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5,10,10, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5,10,10, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,13,13,13,13,13, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,13,13,13,13,13, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,13,13,13,13,13, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,13,13,13,13,13, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,13,13,13,13,13, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
};
```

Although we used both of these arrays in the last chapter, they will take on a slightly different meaning in the block-aligned heightmapped version of the *draw_maze()* function. The *flor[]* array values will not only indicate which tiles are to be mapped onto the corresponding sections of floor, but which tiles will be mapped onto the tops of walls (which are now simply raised sections of floor). The *wall[]* array values now tell us what tiles will be mapped onto the sides of

raised floor sections. In some cases, these "half-height walls" may be only a few pixels high, so only a small portion of the tile will actually be mapped onto them. In other cases these walls may soar as much as 256 pixels above the surrounding floor, requiring tiles larger than the 64-pixel-tall tiles we used in the last chapter. To accommodate such walls, we could include larger tiles in our tile set (though it would be difficult to fit a 256-pixel-tall tile into a 320 by 200 PCX file), or we could write the wall-texturing code so that it starts again at the top of the tile when it reaches the bottom. In this chapter we'll do neither. In the one case where we use a wall that rises more than 64 pixels above the surrounding floor, we'll simply let the wall-texturing routine continue into the tile positioned vertically beneath the wall tile indicated in the *wall[]* array, to provide a vivid illustration of the way in which the wall-texturing routine works.

(Note that a wall that rises *n* pixels above the surrounding floor is not necessarily the same thing as a wall *n* pixels tall. If a wall that is, say, 128 pixels tall is surrounded by floor squares that are 96 pixels tall, then only the top 32 pixels of the wall will need to be texture mapped, not the full 128 pixels. The rest will be hidden by the surrounding floor.)

Heightcasting

Casting for raised sections of floor in a block-aligned heightmapped maze — a process that might be called "heightcasting" — is no different from casting for walls in a fixed-floor-height maze such as the one in the last chapter. So we'll develop our heightcasting routines by taking the wallcasting code from the last chapter and altering it so that it casts for changes in floor height instead of walls. The code for drawing walls and floors will need to be altered too, but we can still use the wall and floor texturing code from the last chapter as a template for creating these parts of the function. Since the majority of the code in the block-aligned heightmapped version of the *draw_maze()* function will be identical to that used in the version of the function we developed in the last chapter, I'll only explain the differences before I show you the function in its entirety.

One difference is in the prototype of the function, which has changed yet again:

```
void draw_maze(map_type wall,map_type floor,
               map_type floorheight,char far *screen,
               int xview,int yview,float viewing_angle,
               int viewer_height,char far *textmaps);
```

The *ceiling* parameter has now been replaced by the *floorheight* parameter, which is a pointer to the *floorheight[]* array. This prototype is in the file BLOKCAST.H in the HEIGHT directory. The function itself is in the file BLOKCAST.CPP, in the same directory.

We'll add a new constant declaration at the head of that file:

```
const GRIDWIDTH=16;
```

The GRIDWIDTH constant tells us the width (and breadth) of the maze in floor squares. We haven't bothered introducing such a constant in earlier chapters since it's only useful in bounds checking, and we've been carefully placing walls around the perimeter of the maze so that bounds checking isn't necessary. With a heightmapped maze, however, placing walls around the boundary of the maze doesn't do us much good, since the wallcasting code will no longer come to a halt when it reaches a wall. In this maze, walls can be visible past other walls (due to height variations), so we'll have to cast until we reach the limits of the maze grid, as defined by GRIDWIDTH. (This need to cast all the way to the edges of the map can slow down the ray casting a bit when the map is large and the viewer is looking toward the more distant end. An obvious optimization is to find a way to cut the casting short when no more walls are going to be visible, though finding such an optimization isn't necessarily a trivial task. Since the heightmapping code won't be used in the optimized ray-casting engine developed in chapter 12, I'll take the lazy way out and leave this optimization as an exercise for the reader.)

How High the Viewer?

Because the floor can now take any height, the first thing the function will need to do is find out how high the floor is directly under the viewer's feet. To find out, we'll calculate the maze grid coordinates on which the viewer is standing by dividing the *xview* and *yview* parameters (which store the viewer's fine coordinates within the maze) by 64:

```
int xmaze=xview/64;
int ymaze=yview/64;
```

We can then consult the *floorheight[]* array to determine the height of this floor square:

```
currentheight=floorheight[xmaze][ymaze];
```

We'll base the code that follows on the assumption that the *viewer_height* parameter passed from the calling routine represents the height of the viewer's eyes above the floor level, though we could just as easily assume that *viewer_height* represents the height of the viewer's eyes above a base level of 0. Since it will also be useful to us to know the viewer's height over such a base level, we'll calculate it by adding *viewer_height* to *currentheight:*

```
int vheight=viewer_height+currentheight;
```

We'll then set up to cast a ray across the maze grid exactly as we did in the last chapter, except that we'll initialize a variable called *lasttop,* the purpose of which will be obvious in a moment:

```
lasttop=VIEWPORT_BOT;
```

Because we'll be casting for walls all the way to the edge of the maze grid, we can no longer use an open-ended *for()* loop. Instead, we'll use a *while()* loop that will watch for either the *xmaze* or the *ymaze* variable (which represent the maze grid coordinates of the next square to be examined) to run off one of the edges of the maze:

```
while((xmaze>=0)&&(xmaze<GRIDWIDTH)&&(ymaze>=0)
            &&(ymaze<GRIDWIDTH)) {
```

We'll also need to nest a second loop inside this one, to perform the actual casting. The outer loop, which we just initialized, will cause this inner loop to be repeated again and again until the edge of the maze is reached. (Remember that we no longer stop casting when we hit a wall.) The inner loop will also be structured as a *while()* loop that will terminate when it reaches the edge of the maze:

```
while((xmaze>=0)&&(xmaze<GRIDWIDTH)&&(ymaze>=0)
       &&(ymaze<GRIDWIDTH)) {
```

(This use of two nested loops with the same terminal condition is not especially efficient and will slow down the code. Readers are encouraged to find ways to make these loops run more efficiently.)

Changes in Floor Height

We then proceed to cast the ray against the x and y grid lines exactly as we did in the last chapter. Instead of looking for walls, though, we'll look for changes in floor height. The current floor height is stored in the integer variable *currentheight,* so all we need to do to find such a change is compare this value with *floorheight[xmaze][ymaze]* for the maze square that we're examining:

```
if (floorheight[xmaze][ymaze] != currentheight) break;
```

We'll need to use this line twice: once in the code that tests for collisions with a wall on an x grid line, and once in the code that tests for collisions with a wall on a y grid line. This will cause the inner loop to terminate when a change in floor height is found.

When such a change is detected, we can find the distance from the viewer to that change exactly as we found the distance to walls in our earlier ray-casting

program. Now, however, the height of the wall will be variable — and the wall may not even be facing toward us. We'll worry about these problems in a moment. For now, we need to determine the y position on the video display at which the bottom of the wall is to be drawn.

We can accomplish this using the similar triangles method that we introduced in the last chapter. (If you don't recall how this method works, you might want to thumb back to that chapter and give yourself a brief refresher course.) In Figure 10-2 you'll see the familiar pair of nested triangles from the last chapter. At that time, we knew the lengths of the lines labeled a, c, and C, and needed to find the length of line A, the distance from the viewer's position to the floor pixel visible through the screen line currently being cast. This time, we know the lengths of A, C, and a, and need to find the length of c, the distance from the center of the display (represented by the constant VIEWPORT_CENTER) to the screen line where the bottom of the wall will appear. The law of similar triangles tells us that the following relationship exists between these lines:

```
c/C = a/A
```

Thus we can find the length of c with the following expression:

```
c = a/A*C
```

In the function that we're developing, the length of *a* is represented by the constant VIEWER_DISTANCE and the length of A by the integer variable

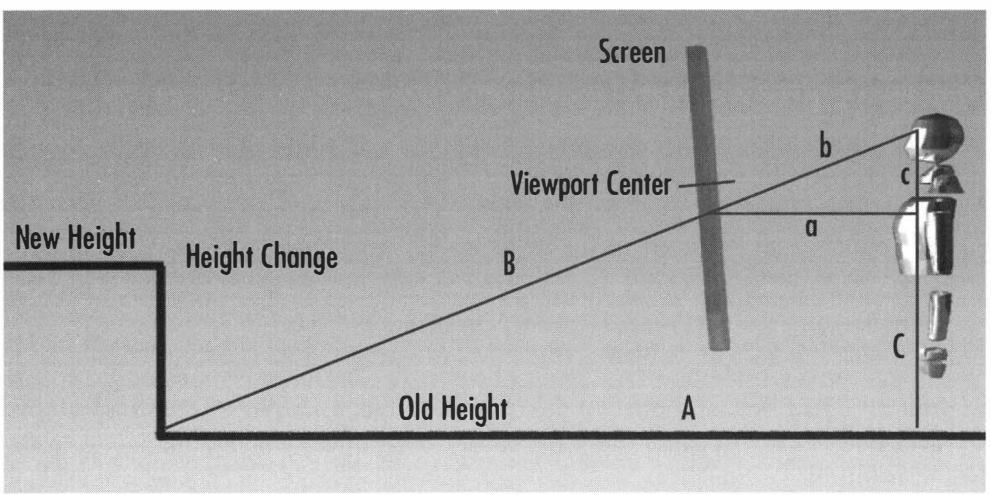


Figure 10-2 The inner triangle, representing the distance from the viewer to the screen, is "similar" in a geometric sense to the outer triangle, representing the distance from the viewer to the wall

distance. Getting the length of C takes slightly more work. The height of the viewer above the floor is contained in the integer variable *viewer_height,* and the height of the viewer above a base level of 0 is contained in the integer variable *vheight*. However, what we really need is the height of the viewer above the floor level at the base of the wall, which is stored in the integer variable *currentheight*. To get the viewer's height above this level, we'll subtract *currentheight* from *vheight*. In the actual function, we'll break the process into two steps. First we'll divide VIEWER_DISTANCE by *distance* and put the result in the floating point variable *ratio:*

```
float ratio=(float)VIEWER_DISTANCE/distance;
```

Then we'll multiply the viewer's height by *ratio* to get the distance from the viewer's eye level to the row of pixels on screen where the bottom of the wall will be drawn and store the result in the integer variable *screendepth:*

```
int screendepth=ratio*(vheight-currentheight);
```

We can translate this into an absolute screen y coordinate by adding *screendepth* to VIEWPORT_CENTER and storing the result in the integer variable *bot:*

```
bot = VIEWPORT_CENTER + screendepth;
```

Before we proceed to draw the wall, we must check first to see if there *is* a wall. If we've reached this point in the code, we know that the ray-casting loop has terminated, but we don't know *why* it terminated. In fact, it can terminate for two different reasons: if a change in floor height has been found or if the edge of the maze grid has been reached. In the latter case, we don't need to draw anything, so we test to make sure we're not past the edge of the maze before we proceed to draw a wall:

```
if ((xmaze>=0)&&(xmaze<GRIDWIDTH)&&
    (ymaze>=0)&&(ymaze<GRIDWIDTH)) {
```

If we aren't at the edge of the maze, then a change in floor height must have been detected. So that we don't have to keep referencing the *floorheight[]* array (which may waste CPU time on array indexing), we'll store the new height of the floor in an integer variable called (what else?) *newheight:*

```
newheight=floorheight[xmaze][ymaze];
```

Next we need to know the height of the wall that we're going to draw — not the visual height, adjusted for perspective, but the actual height in height units. This is simply the difference between the height of the floor just before the change (stored in *currentheight)* and the new height of the floor (which we just stored in *newheight).* We'll store the height of the wall in the integer variable *wallheight:*

```
wallheight=newheight-currentheight;
```

There are two ways in which the height of the floor can change from one floor square to the next, in the direction that the ray is traveling: It can go up or it can go down. If it goes up, then the wall is facing us and we need to draw it. If it goes down, then the wall is facing away from us, and we don't need to draw it (see Figures 10-3a and b). So before we can do anything else, we need to check to see if the value of *wallheight* is positive (meaning that the height of the floor has gone up) or negative. If it is negative, we need to set the value of an integer variable called *top* to be equal to the value of *bot*. We'll see why in a moment.

```
if (wallheight<0) top=bot;
```

If *wallheight* isn't negative, we draw the wall:

```
else { \\ Draw the wall
```

The code for drawing the wall is much like the texture-mapped wall-drawing code that we developed in the last chapter, except that the height of the wall is now variable. We begin by calculating the perspective-adjusted height of the wall and storing it in the integer variable *visheight:*

```
int visheight = (float)VIEWER_DISTANCE * wallheight / distance;
```

We can find the *y* position on screen of the top pixel in the wall by subtracting *visheight* from *bot* and storing the result in an integer variable called *top:*

```
top = bot - visheight;
```

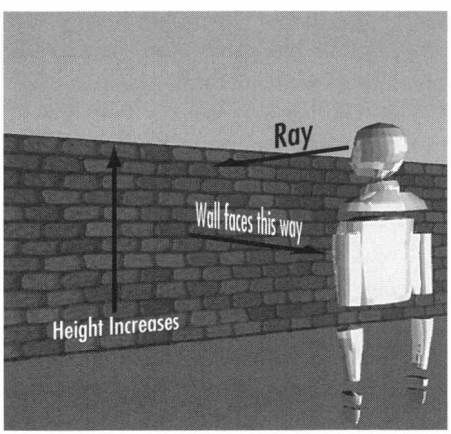


Figure 10-3a If the floor height increases in the direction that the ray is traveling, the wall formed by the raised surface is facing toward the viewer

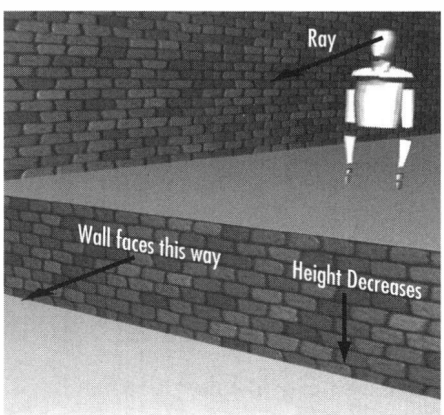


Figure 10-3b If the floor height decreases in the direction that the ray is traveling, the wall formed by the lowered surface is facing away from the viewer

We subtract *visheight* from *bot* rather than adding it because y coordinates on the screen grow smaller as we move upward. Thus, the y coordinate of the top of the wall is lower than the y coordinate of the bottom of the wall.

Before we can draw the wall, we must clip it. In the previous chapter, we clipped the top and bottom of the wall against the top and bottom of the viewport. Now, however, we must also clip the bottom of the wall against the top of any walls that lie between it and the viewer, since these walls would obscure the viewer's view of the wall (see Figure 10-4). Instead of clipping *bot* against VIEWPORT_BOTTOM, as we did in the last chapter, we'll clip it against the integer variable *lasttop,* which we set equal to VIEWPORT_BOTTOM at the beginning of the function. When we finish drawing the wall, however, we'll reset *lasttop* to the *y* position of the top of the wall, so that any walls further along the path of the ray will be clipped against the top of this wall (or any taller walls encountered in the interim).

(Actually, this isn't precisely what will happen. Although we need to save the y coordinate of the top of the wall in *lasttop* for other reasons, we'll actually need to clip the next wall against the top of the backfacing wall on the *other* side of the

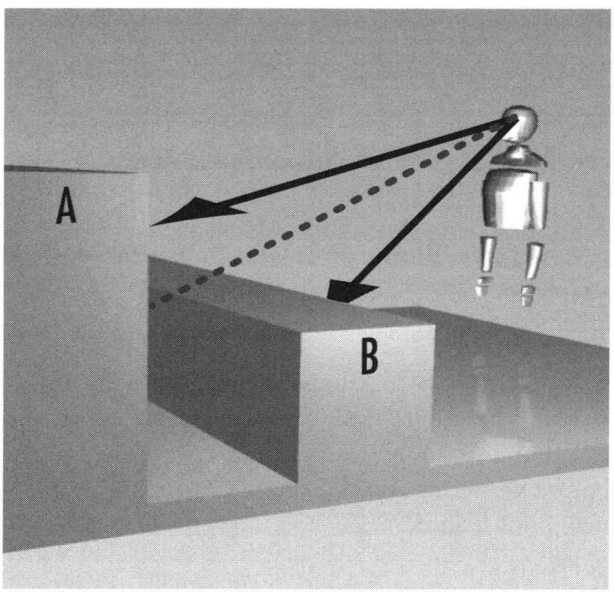


Figure 10-4 The raised floor square at position A obscures the viewer's view of the bottom of the wall created by the raised floor square at position B. Thus we must clip the bottom of wall B against the highest point in floor square A

elevated floor square, as you can see from a close look at Figure 10-4; note the dotted lines. That's why we included instructions earlier that will save the top y coordinate of backward-facing walls in the variable *top,* should *wallheight* be negative. Although backward-facing walls aren't drawn, the y coordinates of their tops will be placed in *lasttop* at the end of the outer casting loop so that more distant walls can be clipped against them.)

The actual drawing of the wall is done exactly as we did it in the last chapter, except that the height of the wall is stored in the variable *wallheight* and only that many pixels from the top of the corresponding texture tile will need to be mapped onto the wall. (As noted earlier, should the wall be taller than the tile assigned to it, the texture mapping code will proceed into the tile positioned beneath it in the PCX file — or into the garbage beyond the end of the file, should the tile be in the bottom row.)

In the last chapter, once the wall was drawn, we drew floor pixels from the bottom of the wall to the bottom of the viewport. We'll do the same thing here, except that instead of drawing to the bottom of the viewport we'll draw floor pixels from the bottom of the wall to the top of the last wall drawn, the y coordinate of which is in *lasttop.* (Initially, as we noted above, *lasttop* will be set to the bottom of the viewport.) We must watch for an additional possibility here, though: The floor pixels may be on an elevated floor square that places them above the viewer's line of sight (see Figure 10-5). If this happens, we must not draw the floor pixels, since the top of the floor square isn't visible. We can determine if this is the case by checking to see if *bot* is below the middle of the viewport (that is, has a y coordinate value greater than VIEWPORT_CENTER):

```
if (bot > VIEWPORT_CENTER) {
```

The only difference in the floor-drawing code from that used in the last chapter is in the beginning of the loop:

```
for (int row=bot+1; row<lasttop; row++) {
```

Now, for reasons explained above, we cast to the top of the last wall drawn rather than all the way to the bottom of the viewport.

Once the floorcasting is done, we only need to change a few variable values before we terminate the outer ray-casting loop. First we set *currentheight* to the new floor height:

```
currentheight=newheight;
```

We also need to set *lasttop* to the height of the top of the wall. However, we must make sure that the current wall isn't a short wall that's completely obscured by a visually taller wall closer to the viewer. In that case, we'll want to leave *lasttop* equal to its previous value:

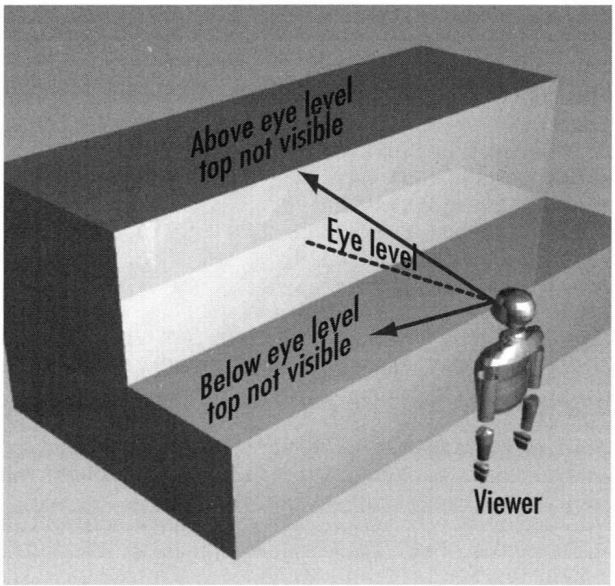


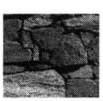
Figure 10-5 If the top of an elevated floor square is above the viewer's line of sight (that is, the center of the screen), the floor pixels on top of the square will not be visible

```
if (top<lasttop) lasttop=top;
```

And that's the end of both the ray-casting loop and the latest version of the *draw_maze()* function.

The Block-Aligned draw_maze() Function

The complete text of the block-aligned heightmapped *draw_maze()* function appears in Listing 10-1.



Listing 10-1 The block-aligned draw_maze() function

```
// BLOKCAST.CPP
//
// Function to draw texture-mapped walls, floors and
// ceilings using ray casting.
// Written by Christopher Lampton for
// Gardens of Imagination (Waite Group Press).
//
```

continued on next page

continued from previous page

```
#include <stdio.h>
#include <math.h>
#include "blokcast.h"
#include "pcx.h"

// Constant definitions:
const WALL_HEIGHT=64;           // Height of wall in pixels
const VIEWER_DISTANCE=192;      // Viewer distance from screen
const VIEWPORT_LEFT=0;          // Dimensions of viewport
const VIEWPORT_RIGHT=319;
const VIEWPORT_TOP=0;
const VIEWPORT_BOT=199;
const VIEWPORT_HEIGHT=VIEWPORT_BOT-VIEWPORT_TOP;
const VIEWPORT_CENTER=VIEWPORT_TOP+VIEWPORT_HEIGHT/2;
const GRIDWIDTH=16;

void draw_maze(map_type wall,map_type floor,
                map_type floorheight,char far *screen,
                int xview,int yview,
                float viewing_angle,int viewer_height,
                char far * textmaps)

// Draws a ray cast image in the viewport of the maze represented
// in array MAP[], as seen from position XVIEW, YVIEW by a
// viewer looking at angle VIEWING_ANGLE where angle 0 is due
// north. (Angles are measured in radians.)

{
  // Variable declarations:
  int sy,offset;       // Pixel y position and offset
  float xd,yd;         // Distance to next wall in x and y
  int grid_x,grid_y;   // Coordinates of x and y grid lines
  float xcross_x,xcross_y; // Ray intersection coordinates
  float ycross_x,ycross_y;
  unsigned int xdist,ydist; // Distance to x and y grid lines
  int distance,realdistance;  // Distance to wall along ray
  int tmcolumn;        // Column in texture map
  float yratio;
  int currentheight; // Height of current floor block
  int newheight,wallheight;
  int heightchange;
  int top,bot,lasttop;
  float x,y;

  // Loop through all columns of pixels in viewport:
  for (int column=VIEWPORT_LEFT; column<VIEWPORT_RIGHT; column++) {

    // Get viewer's initial coordinates:
    int xmaze=xview/64;
    int ymaze=yview/64;

    // Get height of floor under viewer:
```

```
currentheight=floorheight[xmaze][ymaze];

// Calculate height of viewer above base level:
int vheight=viewer_height+currentheight;

// Calculate horizontal angle of ray relative to
//  center ray:
float column_angle=atan((float)(column-160)
                    / VIEWER_DISTANCE);

// Calculate angle of ray relative to maze coordinates
float radians=viewing_angle+column_angle;

// Rotate endpoint of ray to viewing angle:
int x2 = -1024 * (sin(radians));
int y2 = 1024 * (cos(radians));

// Translate relative to viewer's position:
x2+=xview;
y2+=yview;

// Initialize ray at viewer's position:
x=xview;
y=yview;

// Find difference in x,y coordinates along ray:
int xdiff=x2-xview;
int ydiff=y2-yview;

// Cheat to avoid divide-by-zero error:
if (xdiff==0) xdiff=1;

// Get slope of ray:
float slope = (float)ydiff/xdiff;

// Cheat (again) to avoid divide-by-zero error:
if (slope==0.0) slope=.0001;

lasttop=VIEWPORT_BOT;

// Cast ray from grid line to grid line:
while((xmaze>=0)&&(xmaze<GRIDWIDTH)&&(ymaze>=0)
           &&(ymaze<GRIDWIDTH)) {

  // Repeat casting loop until edge of maze is found:
  while((xmaze>=0)&&(xmaze<GRIDWIDTH)&&(ymaze>=0)
         &&(ymaze<GRIDWIDTH)) {

    // If ray direction positive in x, get next x grid line:
    if (xdiff>0) grid_x=((int)x & 0xffc0)+64;

    // If ray direction negative in x, get last x grid line:
```

continued on next page

continued from previous page

```
        else grid_x=((int)x & 0xffc0) - 1;

        // If ray direction positive in y, get next y grid line:
        if (ydiff>0) grid_y=((int)y & 0xffc0) +64;

        // If ray direction negative in y, get last y grid line:
        else grid_y=((int)y & 0xffc0) - 1;

        // Get x,y coordinates where ray crosses x grid line:
        xcross_x=grid_x;
        xcross_y=y+slope*(grid_x-x);

        // Get x,y coordinates where ray crosses y grid line:
        ycross_x=x+(grid_y-y)/slope;
        ycross_y=grid_y;

        // Get distance to x grid line:
        xd=xcross_x-x;
        yd=xcross_y-y;
        xdist=sqrt(xd*xd+yd*yd);

        // Get distance to y grid line:
        xd=ycross_x-x;
        yd=ycross_y-y;
        ydist=sqrt(xd*xd+yd*yd);

        // If x grid line is closer...
        if (xdist<ydist) {

          // Calculate maze grid coordinates of square:
          xmaze=xcross_x/64;
          ymaze=xcross_y/64;

          // Set x and y to point of ray intersection:
          x=xcross_x;
          y=xcross_y;

          // Find relevant column of texture map:
          tmcolumn = (int)y & 0x3f;

          // Is there a maze cube here? If so, stop looping:
          if (floorheight[xmaze][ymaze] != currentheight) break;
        }
        else { // If y grid line is closer:

          // Calculate maze grid coordinates of square:
          xmaze=ycross_x/64;
          ymaze=ycross_y/64;

          // Set x and y to point of ray intersection:
          x=ycross_x;
          y=ycross_y;
```

```
    // Find relevant column of texture map:
    tmcolumn = (int)x & 0x3f;

    // Is there a maze cube here? If so, stop looping:
    if (floorheight[xmaze][ymaze] != currentheight) break;
  }

} // End of inner casting loop

// Get distance from viewer to intersection point:
xd=x-xview;
yd=y-yview;
realdistance=(long)sqrt(xd*xd+yd*yd);
distance=(long)realdistance*cos(column_angle);
if (distance==0) distance=1;

// Calculate bottom of wall on screen:
float ratio=(float)VIEWER_DISTANCE/distance;
int screendepth=ratio*(vheight-currentheight);
bot = VIEWPORT_CENTER + screendepth;

if ((xmaze>=0)&&(xmaze<GRIDWIDTH)&&
    (ymaze>=0)&&(ymaze<GRIDWIDTH)) {

  newheight=floorheight[xmaze][ymaze];
  wallheight=newheight-currentheight;

  if (wallheight<=0) top=bot;
  else {

    // Calculate visible height of wall:
    int visheight = (float)VIEWER_DISTANCE * wallheight / distance;

    // Calculate top of wall on screen:
    top = bot - visheight;

    // Initialize temporary offset into texture map:
    int t=tmcolumn;

    // If top of current vertical line is outside of
    // viewport, clip it:
    int dheight=visheight;
    int iheight=wallheight;
    yratio=(float)wallheight/visheight;
    if (top < VIEWPORT_TOP) {
      dheight-=(VIEWPORT_TOP - top);
      t+=(int)((VIEWPORT_TOP-top)*yratio)*320;
      iheight -= ((VIEWPORT_TOP-top)*yratio);
      top=VIEWPORT_TOP;
  }
  if (bot > lasttop) {
```

continued on next page

continued from previous page

```
            dheight -= (bot - lasttop);
            iheight -= (bot - lasttop)*yratio;
            bot=lasttop;
          }

          // Point to video memory offset for top of line:
          offset = top * 320 + column;

          // Initialize vertical error term for texture map:
          int tyerror=iheight;

          // Which graphics tile are we using?
          int tile=wall[xmaze][ymaze]-1;

          // Find offset of tile and column in bitmap:
          unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
            *IMAGE_WIDTH+t;

          // Loop through the pixels in the current vertical
          // line, advancing OFFSET to the next row of pixels
          // after each pixel is drawn.
          for (int h=0; h<iheight; h++) {

            // Are we ready to draw a pixel?
            while (tyerror>=iheight) {

                  // If so, draw it:
                  screen[offset]=textmaps[tileptr];

                  // Reset error term:
                  tyerror-=iheight;

                  // And advance OFFSET to next screen line:
                  offset+=320;
            }

            // Incremental division:
            tyerror+=dheight;

            // Advance TILEPTR to next line of bitmap:
            tileptr+=320;

          } // End of for() loop

        } // End of if

      } // End of if

      if (bot > VIEWPORT_CENTER) {

        // Step through floor pixels:
```

```
        for (int row=bot+1; row<lasttop; row++) {

          // Get ratio of viewer's height to pixel height:
          float ratio=(float)(vheight-currentheight)
                       / (row-VIEWPORT_CENTER);

          // Get distance to visible pixel:
          distance=ratio*VIEWER_DISTANCE/cos(column_angle);

          // Rotate distance to ray angle:
          int px = - distance * (sin(radians));
          int py = distance * (cos(radians));

          // Translate relative to viewer coordinates:
          px+=xview;
          py+=yview;

          // Get maze square intersected by ray:
          int xmazes = px / 64;
          int ymazes = py / 64;

          // Find relevant column of texture map:
          int t = ((int)py & 0x3f) * 320 + ((int)px & 0x3f);

          // Which graphics tile are we using?
          int tile=floor[xmazes][ymazes];

          // Find offset of tile and column in bitmap:
          unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
            *IMAGE_WIDTH+t;

          // Calculate video offset of floor pixel:
          offset=row*320+column;

          // Draw pixel:
          screen[offset]=textmaps[tileptr];

        } // End of floor loop

      } // End of if()

      currentheight=newheight;
      if (top<lasttop) lasttop=top;
      if (lasttop>VIEWPORT_BOT) lasttop=VIEWPORT_BOT;

    }  // End of outer casting while() loop

  } // End of column for() loop

}
```

The BLOKDEMO Program

To demonstrate this latest maze-drawing function, we'll write a short calling program called BLOKDEMO that will use *draw_maze()* to draw a view of a heightmapped maze. This calling program will use the three arrays shown earlier in this chapter. As in the demo programs in the last chapter, it will read a parameter from the command line that represents the number of radians relative to due north that the viewer should be facing. It will also check for a second parameter representing the height of the viewer above the floor, so that you can look at the maze from both high and low vantage points. This height should be in the range 1 to 255, though you can try other values to see what sort of results you can obtain.

The text of the BLOKDEMO program appears in Listing 10-2.

Listing 10-2 The BLOKDEMO.CPP program

```
// BLOKDEMO.CPP
//
// Calls ray-casting function to draw view of
// block-aligned height fields.
//
// Written by Christopher Lampton for
// Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "screen.h"
#include  "pcx.h"
#include  "blokcast.h"
//#include  "bitmap.h"

const NUM_IMAGES=15;

pcx_struct textmaps,highmaps;

map_type wall={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5,14,14, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 {14, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
```

```
 {14, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 {14, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5,14,14, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,12,12,12,12,12, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,12,12,12,12,12, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,12,12,12,12,12, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,12,12,12,12,12, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,12,12,12,12,12, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
};

map_type flor={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5,10,10, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5,10,10, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5,10,10, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,13,13,13,13,13, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,13,13,13,13,13, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,13,13,13,13,13, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,13,13,13,13,13, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5,13,13,13,13,13, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
};

map_type floorheight={
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20, 0, 0,20,20,20,20,20,20,20,40,60,80,100,120,120},
 {20, 0, 0,20,20,20,20,20,20,20,20,20,20,20,120,120},
 {20, 0, 0,20,20,20,20,20,20,20,20,20,20,20,120,120},
 {20,20,20,20,20,20,20,20,20,20,40,60,80,100,120,120},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20},
 {20,20,20,20,60,60,60,60,60,20,20,20,20,20,20,20},
 {20,20,20,20,60,100,100,100,60,20,20,20,20,40,20,20},
 {20,20,20,20,60,100,150,100,60,20,20,20,40,40,20,20},
 {20,20,20,20,60,100,100,100,60,20,20,40,40,40,20,20},
 {20,20,20,20,60,60,60,60,60,20,20,20,20,20,20,20},
 {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20}
};

float viewing_angle=0;
int viewer_height=32;
int xview=6*64+48;
```

continued on next page

continued from previous page

```
int yview=6*64;

void main(int argc,char* argv[])
{
  // Read arguments from command line if present:
     if (argc>=2) viewing_angle=atof(argv[1]);
     if (argc>=3) viewer_height=atof(argv[2]);

  // Load texture map images:
     if (loadPCX("images.pcx",&textmaps)) exit(1);
     if (loadPCX("highmaps.pcx",&highmaps)) exit(1);

  // Point variable at video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Set mode 13h:
  setmode(0x13);

  // Set the palette:
  setpalette(textmaps.palette);

  // Clear the screen:
  cls(screen);

  // Draw a ray-cast view of the maze:
  draw_maze(wall,flor,floorheight,screen,
              xview,yview,viewing_angle,viewer_height,
              textmaps.image);

  // Wait for user to hit a key:
  while (!kbhit());

  // Release memory
  delete textmaps.image;
  delete highmaps.image;

  // Reset video mode and exit:
  setmode(oldmode);
}
```

To run the program, go to the HEIGHT directory and type BLOKDEMO. If you don't type any parameters on the command line, the view will default to 0 radians and a height of 32 units. You'll see the image depicted in Figure 10-6: a pair of staircases leading up to a platform. Want to see the platform from a higher angle? Hit a key to end the program and type BLOKDEMO 0 150. This will elevate your viewpoint to 150 height units above the floor (which is already 20 units tall), as shown in Figure 10-7.

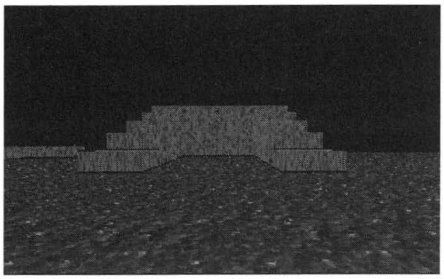


Figure 10-6 The view due north from the center of the heightmapped maze

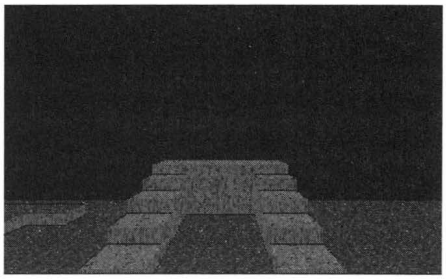


Figure 10-7 The view from 150 height units above the floor

Remember that swimming pool I mentioned earlier? It's right behind you. Do an about face by typing BLOKDEMO 3.14 64 to take a look at it (or just check out Figure 10-8). Another point of interest is the pyramid-like structure to the west. Take a look at it by typing BLOKDEMO 4.7 192 (see Figure 10-9).

Now you're on your own. Experiment with various angles and heights. (Don't overlook that structure in the northwest corner that looks vaguely like the Batplane.) When you get tired of looking around, come back and read the rest of this chapter, where we'll discuss methods for making heightmapping even more versatile and useful.

Heightmapped Ceilings

If you'd like to extend this system, an obvious place to start is by heightmapping the ceilings as well as the floors. Just as the floor in a heightmapped maze can go up, so the ceiling can come down. This can be used to produce interesting effects, such as stalactite-like descending structures and skylights (which would simply be raised portions of ceiling with brightly colored graphics tiles mapped onto them).

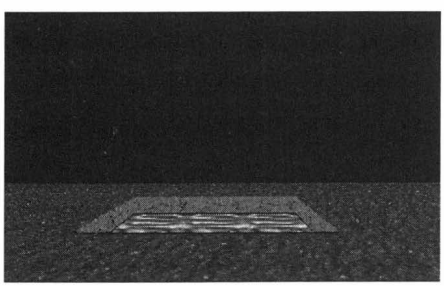


Figure 10-8 Not many mazes have their own swimming pools

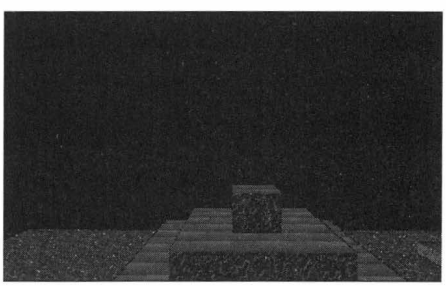


Figure 10-9 The pyramid viewed from a lofty perch

One problem this introduces is hidden surface removal. You must take care not to draw a distant portion of raised floor that would be blocked by a nearer section of lowered ceiling, or vice versa. This means that you'll need to draw the floor and ceiling together, checking each block of floor and the block of ceiling above it for height changes and drawing those changes before moving on to more distant blocks. You'll also need to be careful that you don't lower the ceiling to a level below that of the floor (though lowering the two to the same level, thus forming a wall, shouldn't be a problem).

Tiled Heightmaps

Block-aligned heightmaps are simple to create, require relatively little memory to store, and can be rendered on the display fairly rapidly, if not quite as rapidly as the single-height maze maps in the last chapter. But block-aligned heightmaps suffer from an obvious limitation: They only allow us to elevate square sections of floor. Thus the walls in a maze rendered with this technique are still orthogonal — aligned with the maze grid — and have the same minimum thickness as the maze cubes we used in earlier chapters. They allow us to create multilevel mazes, but they don't free us from the tyranny of the maze grid.

The ideal heightmap would not be block-aligned but pixel-aligned — that is, it would allow every floor pixel to have a different height from the pixels around it. With such a heightmap, we could create mazes containing three-dimensional structures in a wide variety of shapes. We could create slender walls aligned at unpredictable angles. We could create winding staircases and narrow passageways.

Such a heightmap is theoretically possible. We could use a two-dimensional byte array in which each element corresponds to one pixel of the floor to store the height values. You can probably anticipate some of the problems with this approach, though. The most glaring problem is the amount of memory that would be required to store such an array. Since it would contain a 1-byte element for every point in the fine coordinate sytem, a 16 by 16 maze such as we've been using in our demos would require 64 x 16 by 64 x 16, or 1 million, bytes of storage. While this isn't entirely out of the question on current generation machines, most of which contain at least 4 to 8 megabytes of RAM, it does seem rather wasteful. And the size of the array would go up rapidly if the size of the maze were increased. A 32 by 32 maze would require 4 megabytes of heightmap storage, and a 64 by 64 array would require 16 megabytes. There's got to be a more efficient way of storing the heightmap data.

There is. We can tile heightmaps exactly as we've been tiling texture maps. The floors of the mazes that we've created in this and the previous chapter have

been texture mapped at the level of the fine coordinate system. Yet, by the simple expedient of repeating the same texture-map tiles over and over again across the floor of the maze, we've reduced the amount of storage necessary for these texture maps to a small portion of that required to store a PCX image. We can do the same thing with heightmaps: Create a few heightmap tiles and repeat them throughout the maze. The amount of storage required will be minimal, yet we'll be able to pull off heightmapping effects that would be impossible using block-aligned heightmaps.

Alas, there are other problems presented by the use of tiled heightmaps. But we'll discuss those problems in the course of developing a demo showing one possible way in which tiled heightmaps might be used in a ray-casting engine.

Creating Height Tiles

Before we can create a tiled heightmapped version of the *draw_maze()* function, we need a method of creating height tiles to use it with. Like the texture-map tiles that we've been using, each height tile will be a 64 by 64 array of byte values. Typing such height data into an array could get tedious, since each array will have 4,096 entries. But there's a simpler way to create height tiles. We can create them exactly as we create texture-map tiles: with a paint program.

How does one go about creating height tiles with a paint program? In the HEIGHT directory, you'll find a PCX file named HIGHMAPS.PCX. Take a look at it using the PCXSHOW program from chapter 2 (see Figure 10-10). Like the PCX files containing our texture-map tiles, this file contains 15 heightmap tiles stored as 64 by 64 pixel images. The colors in which these tiles have been painted aren't arbitrary. The palette number of each color represents the height of the floor pixel corresponding to that position in the heightmap. For instance, a pixel in a heightmap tile drawn in palette color 47 represents a floor pixel that is 47 height units above the base floor level. Because our ray-casting engine normally sees the

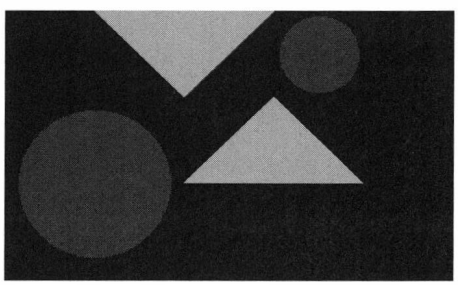


Figure 10-10 The heightmap tiles in the HIGHMAPS.PCX file

pixel values in a bitmap as unsigned byte values, it won't even be necessary to translate the heightmaps into another format before using them. We'll simply store them as a PCX image and look up individual height values the same way we've been looking up individual pixel color values in our texture-map tiles.

While the relative heights of pixels in the heightmap aren't obvious from looking at the PCX image, I've included slanted walls and even curved walls in this PCX file, to demonstrate the flexibility of tiled heightmapping.

The Tiled floorheight[] Array

We'll need to incorporate several arrays into the program to tell the ray-casting engine how to use these height tiles. The first, and most obvious, of these arrays will be the *floorheight[]* array. In our block-aligned heightmapping engine, we used the *floorheight[]* array to store the heights of the floor squares. In this tiled version, we'll use the *floorheight[]* array to store the numbers of the heightmap tiles to be used for the floor squares. A value of 5 in the *floorheight[]* array, for instance, means that the heightmap for the corresponding floor square is in height tile number 5. You'll note that the *floorheight[]* array serves the same purpose for heightmap tiles as our *flor[]* and *wall[]* arrays have served for texture-map tiles. Here's the *floorheight[]* array that we'll be using in our demonstration program:

```
map_type floorheight={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 3, 3, 3, 3, 3, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 5,10, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6,11, 0, 0},
 { 0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};
```

The floorbase[] Array

We'll also create an array called *floorbase[]*, the purpose of which may not be readily apparent. Essentially, the *floorbase[]* array will work exactly like the

floorheight[] array in the block-aligned heightmapped engine. The numbers in the *floorbase[]* array will specify the heights of the individual maze squares. In the last program, we calculated the height of the floor from an assumed base of 0. Now we'll give each floor square an individual base value, as specified in the floor base array. The actual heights of the individual floor pixels will be calculated as the sum of the value for that square in the *floorbase[]* array and the value for that pixel in the heightmap tile corresponding to that square. What's the point of maintaining a *floorbase[]* array? It will allow us to reuse height tiles at various height levels throughout the maze. Despite the versatility of the tiled heightmapping approach, most of the squares in a maze will still be flat. But some flat squares will be at low levels within the maze while others may be on top of towering balconies. There's no point in creating several flat height tiles when we can simply raise the floor base and use the same flat tile at each level. Similarly, tiles that represent walls rising from a flat floor can be used in both low-lying portions of the maze and elevated portions of the maze, simply by raising the floor base underneath the tiles. The floorbase array that we'll be using in our demo program looks like this:

```
map_type floorbase={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,40,20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,40,40,20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,40,20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};
```

The Tiled flor[] and wall[] Arrays

Finally, we'll maintain the *flor[]* and *wall[]* arrays from earlier versions of our ray-casting engine. The *flor[]* array will store indices into a PCX file filled with texture maps, as it did in our earlier programs, but the meaning of the *wall[]* array will change. Previously, the *wall[]* array was used to store pointers to the texture-map tiles that were to be mapped onto the corresponding sections of wall. Now that we have the ability to create walls in any shape and oriented at any

angle, texture mapping is considerably more difficult. When the walls were guaranteed to be aligned with the maze grids, we could find the appropriate column of the texture map by taking the x or y coordinate at which the ray struck the wall modulo 64. But how do we find the appropriate column of the texture map to map onto a diagonal wall? Or a curved wall? In this chapter we'll take the easy way out and drop the texture-mapped walls altogether, drawing the walls in solid colors instead. The number in the *wall[]* array for a given square will indicate what color the wall is to be drawn in. This allows us to perform a kind of rough-and-ready shading on straight and diagonal walls, but it doesn't work very well on curved walls, as you'll see in a moment. After we demonstrate the tiled heightmapping engine, I'll suggest some alternate methods that might be used for texture mapping heightmapped walls.

Now let's develop a tiled heightmapping version of the *draw_maze()* function.

A Tiled Heightmapping Engine

Because height changes in a tiled heightmap are aligned at the level of individual floor pixels rather than at the level of the maze grid, we can no longer cast for walls by casting rays against x and y grid lines. The best way to cast for height changes would, in fact, be to test every pixel along the path of the ray to see if a height change has occurred, perhaps using a variation on Bresenham's algorithm to determine the path that the ray takes. Such a program wouldn't be terribly difficult to write, but it would be excruciatingly slow in execution.

An alternative method would be to cast for height changes at the same time as we cast for the color of the floor pixels. This would not only be relatively fast, but it would be easy to write. We could drop the wallcasting code altogether and write a tight floorcasting loop instead. The disadvantage of this method is that it can miss changes in floor height altogether, especially when the ray is more than a few maze squares away from the viewer and the angle of the ray becomes shallow. For the most part, this won't matter. If a wall or raised section of floor is thick enough, we'll pick it up a few floor pixels past the actual height change. The fact that the ray penetrates a slight distance into the wall won't be obvious on the screen. There's a second drawback to this method as well, but we'll talk about that later in the chapter.

We'll use the second method in this chapter, not because it's perfect but because it's relatively fast. A few years from now, about the time that Intel introduces the Octium microprocessor, the average PC should be fast enough to make the first method feasible.

Here's the prototype for this latest version of the *draw_maze()* function:

```
void draw_maze(map_type wall,map_type floor,
               map_type floorheight,map_type floorbase,
```

```
char far *screen,int xview,int yview,
float viewing_angle,int viewer_height,
char far *textmaps,char far *highmaps)
```

This version passes the address of the buffer containing the heightmaps in the pointer *highmaps* and a pointer to the array of floorbase heights in *floorbase*. We'll place this prototype in the file TILEDEMO.H in the HEIGHT directory. The function will go in the TILEDEMO.CPP file in the same directory.

This version of the *draw_maze()* function, like all of the other ray-casting versions, is structured as a large *for()* loop that iterates through all of the columns in the viewport. At the beginning of the loop, we calculate the viewing angle for the current column exactly as in earlier versions. Then we calculate the viewer's height much as we did in the previous function, by determining which maze square the viewer is standing on and retrieving the corresponding value from the *floorheight[]:*

```
int xmaze = xview/64;
int ymaze = yview/64;
int floortile=floorheight[xmaze][ymaze];
```

This time, however, we use a somewhat different method of retrieving the height of the pixel that the viewer is standing on, employing the same method that we've used in the past for finding the color of a floor pixel in a texture-map tile. Remember that the heightmaps, like the texture maps, are 64 by 64 tiles in a 320 by 200 PCX image, arranged in three rows of five maps each. We can find the number of the row by dividing the tile number (in *floortile)* by 5, get the size of a row by multiplying 320 (the screen width) by 64 (IMAGE_HEIGHT), and then multiply the tile number by the row size. To get the position of the tile, we take the tile number modulo 5, then derive the row offset by multiplying this number by 64 (IMAGE_WIDTH). Finally, we obtain the actual position within the tile by taking *yview* modulo 64 (IMAGE_HEIGHT), multiplying it by 320 (the screen width), and adding in *xview* modulo 64 (IMAGE_WIDTH). Whew! Here's the resulting instruction:

```
int currentheight=highmaps[(floortile/5)*320*IMAGE_HEIGHT
            +(floortile%5)*IMAGE_WIDTH
            +(yview%IMAGE_HEIGHT)*320
            +(xview%IMAGE_WIDTH)];
```

We'll cast for the floor pixels exactly as we did in the ray-casting engine we developed in the last chapter. However, we'll include code in the floorcasting loop to abort the casting when the edge of the map is reached:

```
if ((xmaze<0)||(xmaze>=GRIDWIDTH)
    ||(ymaze<0)||(ymaze>=GRIDWIDTH))
  break;
```

As in the first program in this chapter, we can no longer depend on there being a wall to stop us at the edge of the maze.

We'll determine the height of floor pixels much as we determine the color of floor pixels. First we'll calculate a value *t* that represents the column in the height tile corresponding to the current floor pixel:

```
int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);
```

Then we'll store the number of the height tile for the current floor pixel in the variable *tile:*

```
int tile=floorheight[xmaze][ymaze];
```

Finally, we'll calculate the offset of the relevant byte of height data within the PCX image containing the heightmaps:

```
unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;
```

To get the actual height of the floor pixel, we need to add the height value from the heightmap file and the *floorbase[]* array value for the maze square containing the floor pixel:

```
int newheight=highmaps[tileptr]+floorbase[xmaze][ymaze];
```

If the floor level has gone down, the wall is facing away from us, so no wall needs to be drawn. All we need to do is reset the floor height and keep casting:

```
if (newheight < currentheight) currentheight=newheight;
```

The other possibilities, of course, are that the floor level has gone up or that it has not changed at all. In the latter case, we do nothing and continue casting. If the floor has gone up, then we prepare for drawing the wall:

```
else {
  if (newheight > currentheight) {
```

We still need to reset the floor height:

```
currentheight=newheight;
```

We can use the similar triangles method to find the screen row corresponding to the top of the wall:

```
ratio = (float)VIEWER_DISTANCE/real_distance;
int a = ratio*(viewer_height-currentheight);
int newrow = VIEWPORT_CENTER + a;
```

We can then calculate the offset in video memory to begin drawing the wall at and draw pixels from there to the screen row corresponding to the bottom of the wall:

```
offset=row*320+column;
for (int i=row; i>newrow; --i) {
  screen[offset]=wall[xmaze][ymaze];
  offset-=320;
}
```

Note that the color for the wall is taken from the *wall* array entry corresponding to the current maze square.

Once the wall is drawn, we can reset the *row* variable to the screen row directly above the top of the wall:

```
  row=newrow;
}
```

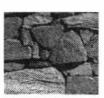
Whether or not there was a change in floor height, we must draw the floor pixel at the current position. However, we must first check to see that the floor pixel is visible — that is, that it is below the viewer's eye level, which is presumed to be at VIEWPORT_CENTER:

```
if (row > VIEWPORT_CENTER) {
```

The rest of the floorcasting — which is the rest of this function — proceeds exactly as it did in the ray-casting engine in the last chapter.

The Tiled Heightmapped draw_maze() Function

The complete text of the tiled heightmapped version of the *draw_maze* function appears in Listing 10-3.



Listing 10-3 The tiled heightmapped draw_maze() function

```
//
// TILECAST.CPP
// Draws a tiled heightmapped three-dimensional maze
// Written by Christopher Lampton
// for Gardens of Imagination (Waite Group Press)

#include <stdio.h>
#include <math.h>
#include "tilecast.h"
#include "pcx.h"

const WALL_HEIGHT=64;           // Height of wall in pixels
const VIEWER_DISTANCE=128;      // Viewer distance from screen
const VIEWPORT_LEFT=0;          // Dimensions of viewport
```

continued on next page

continued from previous page

```
const VIEWPORT_RIGHT=319;
const VIEWPORT_TOP=0;
const VIEWPORT_BOT=199;
const VIEWPORT_HEIGHT=VIEWPORT_BOT-VIEWPORT_TOP;
const VIEWPORT_CENTER=VIEWPORT_TOP+VIEWPORT_HEIGHT/2;
const ZOOM=2;
const GRIDWIDTH=16;
const WALLCOLOR=18;

void draw_maze(map_type wall,map_type floor,
                map_type hightile,map_type floorbase,
                char far *screen,int xview,int yview,
                float viewing_angle,int viewer_height,
                char far *textmaps,char far *highmaps)
{
  int y_unit,x_unit; // Variables for amount of change
                     //  in x and y
  int distance,real_distance,old_distance,xd,yd,sx,sy;
  unsigned int offset;

  // Loop through all columns of pixels in viewport:
  for (int column=VIEWPORT_LEFT; column<VIEWPORT_RIGHT; column++) {

    // Calculate horizontal angle of ray relative to
    //  center ray:
    float column_angle=atan((float)((column-160)/ZOOM)
                        / VIEWER_DISTANCE);

    // Fudge column angle:
    if (column_angle==0.0) column_angle=0.0001;

    // Calculate angle of ray relative to maze coordinates
    float radians=viewing_angle+column_angle;

    // Which square is the viewer standing on?
    int xmaze = xview/64;
    int ymaze = yview/64;

    // Get pointer to floor height map:
    int floortile=hightile[xmaze][ymaze];

    // How high is the floor under the viewer?
    int currentheight=highmaps[(floortile/5)*320*IMAGE_HEIGHT
                +(floortile%5)*IMAGE_WIDTH
                +(yview%IMAGE_HEIGHT)*320
                +(xview%IMAGE_WIDTH)];

    // First screen row to draw:
    int row=VIEWPORT_BOT;

    // Cast a ray across the floor:
    for (;;) {

      // Get ratio of viewer's height to pixel height:
```

```
float screen_height=row-VIEWPORT_CENTER;
if (screen_height==0.0) screen_height=.00001;
float ratio=(float)(viewer_height-currentheight)/screen_height;

// Get distance to pixel:
real_distance=ratio*VIEWER_DISTANCE;
if (real_distance>old_distance+5) distance+=5;
distance=real_distance/cos(column_angle);

// Rotate distance to ray angle:
int x = - distance * (sin(radians));
int y = distance * (cos(radians));

// Translate relative to viewer coordinates:
x+=xview;
y+=yview;

// Get maze square intersected by ray:
int xmaze = x / 64;
int ymaze = y / 64;

// If we've reached the edge of the map, quit:
if ((xmaze<0)||(xmaze>=GRIDWIDTH)
    ||(ymaze<0)||(ymaze>=GRIDWIDTH))
  break;

// Find relevant column of texture and height maps:
int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);

// Which height tile are we using?
int tile=hightile[xmaze][ymaze];

// Find offset of tile and column in bitmap:
unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;

// Get height of pixel:
int newheight=highmaps[tileptr]+floorbase[xmaze][ymaze];

// Has the floor level gone down?
if (newheight < currentheight) currentheight=newheight;

else {

  // Has the floor level gone up?
  if (newheight > currentheight) {

    // If so, set new floor height:
    currentheight=newheight;

    // If so, calculate new screen position:
    ratio = (float)VIEWER_DISTANCE/real_distance;
```

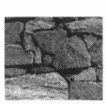
continued on next page

continued from previous page

```
            int a = ratio*(viewer_height-currentheight);
            int newrow = VIEWPORT_CENTER + a;

            // Draw wall segment:
            offset=row*320+column;
            for (int i=row; i>newrow; --i) {
              screen[offset]=wall[xmaze][ymaze];
              offset-=320;
            }

            // Set screen row to new position:
            row=newrow;
          }

          // If viewer is looking down on floor, draw floor pixel:
          if (row > VIEWPORT_CENTER) {

            // Which graphics tile are we using?
            tile=floor[xmaze][ymaze];

            // Find offset of tile and column in bitmap:
            tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)*IMAGE_WIDTH+t;

            // Calculate video offset of floor pixel:
            offset=row*320+column;

            // Draw pixel:
            screen[offset]=textmaps[tileptr];

            // Advance to next screen line:
            row--;
          }
        }
      }
    }
}
```

The TILEDEMO.CPP Program

To demonstrate this new ray-casting function, we'll call it from a short program called TILEDEMO.CPP. The complete text of this program is in Listing 10-4.



Listing 10-4 The TILEDEMO.CPP program

```
// TILEDEMO.CPP
//
// Calls ray-casting function to draw view of
```

```
// tiled height-mapped maze.
//
// Written by Christopher Lampton for
// Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "screen.h"
#include  "pcx.h"
#include  "tilecast.h"

const NUM_IMAGES=15;

pcx_struct textmaps,highmaps;

map_type wall={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,28, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,28,28, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,28,28,28, 0, 0, 0, 0, 0, 0, 0, 0,18,18, 0, 0},
 { 0,22,22,22, 0, 0, 0, 0, 0, 0, 0, 0,18,18, 0, 0},
 { 0,17,17,17, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,17,17, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,17, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};

map_type flor={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 3, 3, 3, 3, 3, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,12, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,12,12, 1, 0, 0, 0, 0, 0, 0, 0, 0, 5,10, 0, 0},
 { 0,12,12,12, 0, 0, 0, 0, 0, 0, 0, 0, 6,11, 0, 0},
 { 0,12,12, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,12, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

continued on next page

continued from previous page

```
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};

map_type floorheight={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 3, 3, 3, 3, 3, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 5,10, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6,11, 0, 0},
 { 0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};

map_type floorbase={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,40,20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,40,40,20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,40,20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};

float viewing_angle=0;
int viewer_height=150;
int xview=8*64+32;
int yview=7*64;

void main(int argc,char* argv[])
{
  // Read arguments from command line if present:
  if (argc>=2) viewing_angle=atof(argv[1]);
  if (argc>=3) viewer_height=atof(argv[2]);
```

```
  // Load texture map images:
  if (loadPCX("images2.pcx",&textmaps)) exit(1);
  if (loadPCX("highmaps.pcx",&highmaps)) exit(1);

  // Point variable at video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Set mode 13h:
  setmode(0x13);

  // Set the palette:
  setpalette(textmaps.palette);

  // Clear the screen:
  cls(screen);

  // Draw a raycast view of the maze:
  draw_maze(wall,flor,floorheight,floorbase,screen,
              xview,yview,viewing_angle,viewer_height,
              textmaps.image,highmaps.image);

  // Wait for user to hit a key:
  while (!kbhit());

  // Release memory
  delete textmaps.image;
  delete highmaps.image;

  // Reset video mode and exit:
  setmode(oldmode);
}
```

To run the program, go to the HEIGHT directory and type TILEDEMO. You'll see the circular formation in Figure 10-11. Type TILEDEMO 3.14 and you'll see the wedge-shaped structure in Figure 10-12. And if you type TILEDEMO 1.57, you'll see the series of circular posts in Figure 10-13. The texture-map tiles for the floor are taken from the files IMAGES2.PCX in the HEIGHT directory. They were created specifically to go with the height tiles in HIGHMAPS.PCX.

Type TILEDEMO 3.14 250 to get a look at the wedge-shaped structure from a considerable height, and you'll see another flaw in this method of casting for height changes (see Figure 10-14). The front of the wedge-shaped structure, which should be partially obscured by the bottom of the viewport, will move back until it's flush with the bottom. Why does it do this? Because we didn't cast for height changes between the viewer and the bottom of the viewport. The

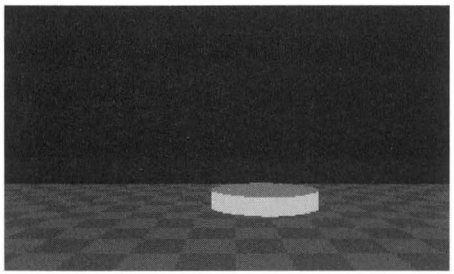


Figure 10-11 A circular formation made from height tiles

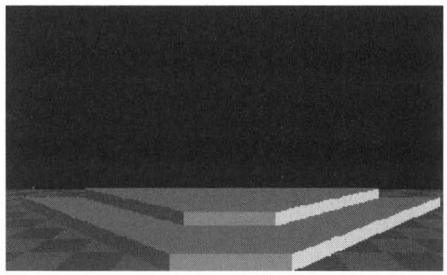


Figure 10-12 A wedge-shaped structure

Figure 10-13 A row of circular posts

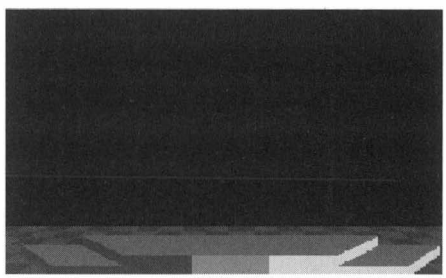


Figure 10-14 Height changes between the viewer and the bottom of the viewport are missed by TILEDEMO

program assumes that no such changes occur. We didn't cast for such height changes because that would slow down the code considerably, and would make the function considerably more complicated to write. Similarly, if any height changes are obscured by a closer wall, they will be missed by the floorcasting routine. One solution to this problem might be to perform spot checks for height changes below the bottom of the viewport and behind walls, but slender walls (or the corners of walls) might still be missed.

Hybrid Heightmapping Methods

Which, if any, of the above methods are programs like Doom using? Probably neither. It's clear from the visual effects that it achieves that Doom is not using block-aligned heightmapping, a method that can provide some quick and dirty heightmapping tricks but is too limited to depict walls that aren't aligned at 90 degrees to the maze grid or that are less than a maze square in thickness. And it probably does not use tiled heightmaps, which have flaws that only the CPU-intensive solutions suggested in the last paragraph would solve.

There are, however, heightmapping algorithms that combine the elements of both techniques. There's not enough space in this book to cover them, but you might want to explore them on your own. Basically, these hybrid methods would specify height changes on a block-aligned basis, so that it would only be necessary to check for height changes at block boundaries. But the height changes themselves would not necessarily have to take place at those boundaries. While only one height change might be allowed per block, that height change could slash across the middle of the block at an arbitrary angle rather than along the edges of the block. Each entry in the *walls[]* array could contain a flag indicating that a height change takes place in that block. If one does, the entry would also contain the slope at which that change is oriented relative to the maze grid and the coordinates at which it crosses the edges of the block (or a pointer to a separate structure that contains this information). A more ambitious system would allow each block to contain a pointer to a linked list that could contain an unlimited number of height changes, all of which occur in that block. Such a system could simulate a tiled heightmap system, but would run much faster.

Is this what the Doom programmers have done? I have no idea. However, it should be possible to duplicate most if not all of the floor and wall effects in Doom by using such a system.

Animated Heightmaps

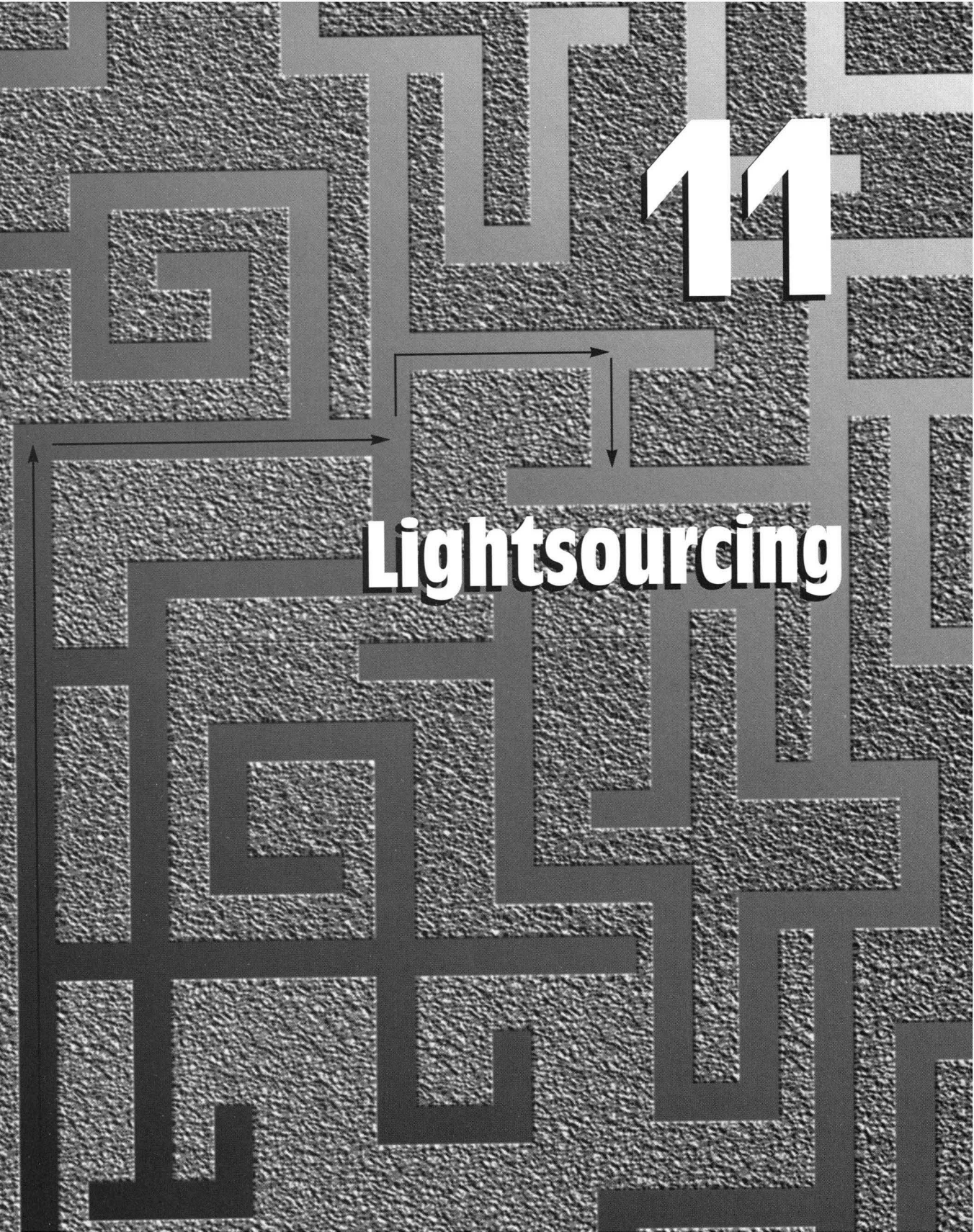
Once you've developed a heightmapping system, there's no reason that you should stop there. There are a number of special effects that can be achieved by *animating* the heightmaps. For instance, you could raise and lower heightmaps dynamically to create an elevator. In a block-aligned heightmap, this is only a matter of changing the number in a single position in the *floorheight[]* array, assuming that the elevator occupies a single maze square. (For larger elevators, just change the numbers in several different positions.) By increasing the floor height on every frame, a section of floor will appear to detach itself and rise toward the ceiling atop a thick column. If the player character is standing on that block, he or she will automatically be lifted along with it. (This will be handled automatically by the ray-casting code in this chapter, which always checks the height of the floor pixels beneath the viewer's feet before casting the image.) Reducing the number in the height array corresponding to a maze square will cause the floor of that square to go down, even below the height of surrounding squares if those squares are higher than the base level.

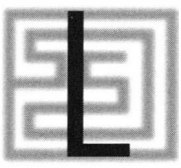
Changing the floor level more rapidly than this creates a trapdoor over a hidden pit. When the unsuspecting player character steps on that square, lower it by a few dozen height levels — and the viewer will seem to plunge in after it. You

can then lower his or her hit points to represent damage from the fall, if that strikes you as appropriate. And you can use texture mapping to write messages on the walls of the pit that the player cannot see until he or she takes the plunge, perhaps representing important clues toward completing the game.

If you're using a tiled heightmapping system, raising or lowering the floor need not be any more complicated than with a block-aligned system. To raise a section of floor, you can change the level in the *floorbase[]* array rather than altering one of the height tiles. If you want your elevator to have nonaligned walls, you can maintain a sequence of height tiles that represent the floor's height at various stages in the process. And if this requires too many height tiles for the available memory in the computer, you can actually modify the height tile itself dynamically.

Once you get into dynamically modifying height tiles, even more bizarre effects become possible. For instance, the floor can morph from one shape to another, going from flat to pyramidal, or sinking to form a bowl-like pit with sloping edges. Use your imagination. With heightmaps, the sky's the limit!


11
Lightsourcing



ight is the soul of ray casting. By following beams of light from the viewer's eye through an array representing a maze, we can produce remarkably realistic visual effects. So far, however, we've only modeled the way in which light moves through the maze. We've made no attempt to capture the way in which light intensities vary across the surfaces in the maze.

This may not sound terribly important, but by giving our ray-casting engine the ability to dynamically alter the brightness of surfaces according to the ways in which rays of light strike them — a process known as *lightsourcing* — we can greatly enhance the realism of the images that it produces. Our murky mazes will really *look* murky, with shadows filling distant hallways and corners of large rooms. Distant surfaces can be made darker than nearer ones, a visual cue that will make our mazes look even more three-dimensional than they already do. Pools of light can be placed around lamps and torches. Flashes of light can accompany the explosion of a fireball. And lightsourcing can also be used to create visual effects that at first glance seem to have little to do with light, such as filling the maze with fog or even water. Best of all, lightsourcing is fairly easy to add to a ray-casting engine and the cost in added CPU cycles is small.

Casting Some Light on Light

Light is a form of radiation. That doesn't mean that you should don a radiation suit the next time you head to the beach (though some SPF-30 sunblock might not hurt). It simply means that light radiates outward in all directions from the source that's producing it, as in Figure 11-1. We can imagine that the wavefront of photons leaving a light source at any given instant in time is an expanding sphere, growing larger at the speed of light (see Figure 11-2). If there's nothing standing in its way, this sphere of light can expand for a very long time. When astronomers turn their telescopes on distant quasars, they're catching a tiny portion of an expanding light sphere that left its source more than ten billion years ago. That's a long time to spend traveling in the icy vacuum of space.

If these particles of light happen to be heading in your direction and you happen to be looking toward the light source when they reach you, some of them will strike your eyes. If enough of them strike at one time, they will activate photosensitive receptors and trigger a visual signal in your brain. This visual signal tells you that you're looking toward a light source. You will, quite literally, see the light.

How bright the light appears depends mostly on three factors: how much light the light source is producing, how much of the light gets absorbed or knocked aside on its way to your eyes, and how far away you are from the source. We'll talk about all of these factors in the course of this chapter, but for now we'll concentrate on the third: distance.

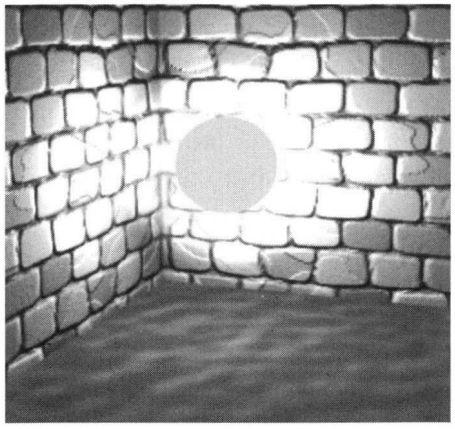


Figure 11-1 Light spreads out in all directions from its source

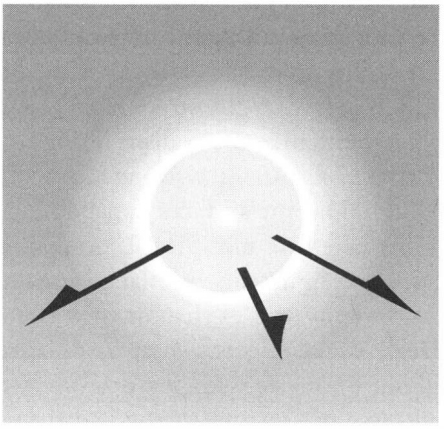


Figure 11-2 The photons leaving a light source at any given instant form an expanding sphere of light around the source

As the sphere of light expands, the particles making up the sphere grow farther apart. When the sphere reaches your eye, fewer particles hit your eye and the visual signal in your brain grows weaker, making the light seem dimmer. This is why stars, the brightest light sources that you can see with your naked eyes, look a lot dimmer than the feeble streetlamps that routinely drown out their light.

Most of the light that reaches our eyes doesn't come directly from a light source. It is reflected off the surfaces of other objects. The amount of light reaching these surfaces also depends on how close they are to the light source. The closer a surface is to a light source, the more light it reflects and the brighter it appears to be. The farther away it is from a light source, the darker the surface appears. This effect won't be obvious when you're outside on a sunny day, since most of the objects that you'll see are roughly equidistant from their primary light source: the sun. But it's immediately apparent in a room that is illuminated by a single light source, such as a lamp. Light will seem to pool in the vicinity of the lamp, but will fall off into dark shadows in distant corners.

What does this have to do with maze games? A lot of games take place in dark, dank dungeons, which are presumably illuminated only by a torch carried in the hand of the player character. Presenting such mazes in full illumination, as we have done up until now, isn't really realistic. In the real world, the player would find that surfaces in his or her immediate vicinity are more brightly illuminated than surfaces in the distance. In a large room or a long corridor, distant walls may be so dimly lit as to not be visible at all. To the player, the light of the torch would seem to form a circular pool of brightness that dims rapidly into the distance (see Figure 11-3).

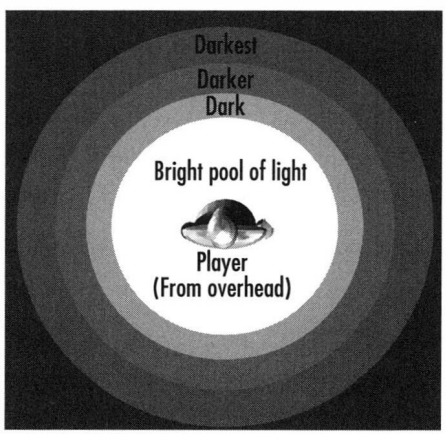


Figure 11-3 The light of the torch forms a circular pool around the player, dimming into the distance

We can simulate this effect in our ray-casting engine by making distant walls darker and by making the pixels in both the floor and ceiling grow dimmer as they recede from the viewer. This addition won't even take a lot of extra code. It will, however, require at least some understanding of how the intensity of light decreases with distance.

The Inverse Square Law

A moment ago, I compared the wavefront of photons leaving a light source to an expanding sphere with the light source at its center. The density of photons on the "surface" of this sphere determines how bright the light source will appear to an observer. Because the photons spread out as the sphere expands, the light source appears dimmer with distance. A little thought tells us that there's a simple rule governing the rate at which the light dims.

Figure 11-4a depicts a light sphere with a radius of r. If you were standing at a distance of r from the light source producing this sphere, your eyes would intercept photons from the surface of this sphere. The small square on the surface of the sphere represents the section of the sphere that strikes, and enters, your eyes. (I've exaggerated the size of this square to make it easier to see.) Figure 11-4b represents a light sphere with a radius of 2 x r, twice that of the sphere in Figure 11-4a. The tiny square has expanded to a square with twice the height and twice the width (and therefore four times the area) of the square in Figure 11-4a. However, if you were standing at a distance of 2 x r from the light source, your eyes — which haven't grown any larger — would only intercept the photons in

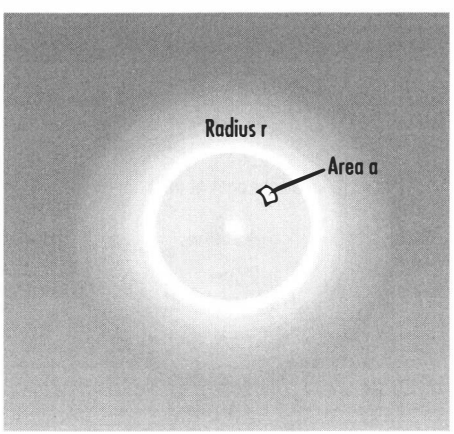


Figure 11-4a An expanding light sphere. The square patch is the area intercepted by your eyes

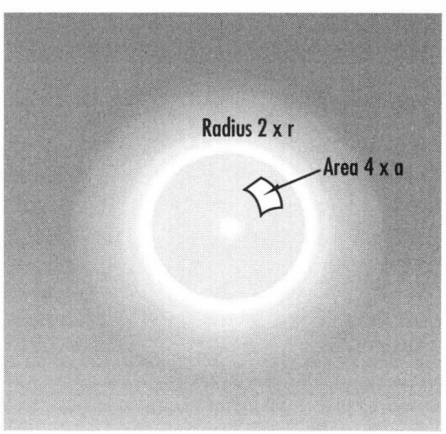


Figure 11-4b The same sphere at twice the distance. The square patch has expanded to four times its previous area

the smaller dotted square inside of the large square. This square contains only one-fourth as many photons as the original, since the photons from the original square have now spread out to form the larger square. Thus the light source would appear only one-fourth as bright.

The same thing would happen if you were standing 3 x r from the light source, except that the small square would only be 1/9th as large as the original square and thus the light would seem 9 times dimmer. At a distance of 4 x r, the small square would be only 1/16 as large as the original square and the light would seem 16 times dimmer. At 5 x r, the small square would be 1/25th as large as the original square and the light 25 times dimmer. And so forth.

Astute readers will notice that the light grows dimmer as the square of the distance increases. This is known as the *inverse square law* and represents the way that the intensity of *all* forms of radiation falls off with distance, not just light. As a general rule, we can say that if a light has an apparent brightness of B at a distance of D, it will have an apparent brightness of $1/(X^2)$ at a distance of X x D.

There's a catch to this rule, however. It only applies to a light source that has no size — that is, it assumes that the light is emanating from a single point in space. In the real world, light sources tend to be larger than this, which causes the light to fall off more slowly with distance. So we can't use the inverse square law as a realistic model for the way light intensity changes. What will we do?

The Lightsourcing Formula

My own solution was to pull a copy of *Computer Graphics* by Donald Hearn and M. Pauline Baker (Prentice Hall, 1986) off my bookshelf and look to see if it contained any formulas relating to light intensity. What I found was this:

$$I = \frac{k_d I_p}{d + d_o}(N \cdot L)$$

where I is the intensity of the light on the surface, k_d is the reflectivity of the surface, I_p is the intensity of the light source, d is the distance from the light source to the surface, d_0 is a fudge factor that avoids divide-by-0 errors when the light source and the surface are *very* close, N is a vector representing the surface normal of the surface, and L is a vector representing the direction of the ray of light striking the surface.

Got that? Good. Now forget it, because we're not going to be using that formula. We don't need the business with the surface normal and the light ray vector because the light source is always originating at the viewer (who will be carrying the torch) and the light always reflects straight back from the surface to

the viewer. Thus the two factors cancel each other out. We don't need the fudge factor because we already have a fudge factor preventing the distance from getting too small. And we don't need the reflectivity factor because we're going to assume that all surfaces in the maze are equally reflective.

In practice, I find that this simpler formula produces perfectly reasonable results:

```
intensity_ratio = intensity_of_source / distance_from_source * MULTIPLIER
```

where *intensity_ratio* is a floating point number in the range 0 to 1 that represents how bright the surface appears relative to how it would appear in full light (with 1 representing full intensity and 0 representing black), and MULTIPLIER is a constant that prevents the light intensity from falling off too rapidly with distance, so that even nearby walls would be invisible. I recommend a value of 3 for this constant, but feel free to try other values.

There's nothing sacred about this formula. I use it because it works. But if you find one that works better, you should use it in preference to this one (and I wouldn't mind if you let me use it too). There's no point in spending too much time worrying about whether our lightsourcing equations produce accurate results, though. What matters is that those results look realistic to the viewer, who probably won't be examining the video display with a light meter. Thus we can get away with a lot of simplification.

Before we can use this formula, we need to know what numbers to assign to the variables. Obviously, the *distance_ from_source* factor in this formula will be the distance of the light source in pixel units from the coordinates *xview,yview* at which the viewer is standing, since our engine already calculates that distance and makes it readily available in the variable *distance*. But how are we going to measure the intensity of the light source? Surprisingly, we can measure it using any method we want to, as long as we use that method consistently throughout the program.

The method that we'll be using in this chapter will represent the intensity of light sources with a number in the range 0 to 32, with 32 representing the maximum possible intensity and 0 representing complete darkness. We'll measure the brightness of surfaces the same way, with a brightness of 32 indicating that the pixels appear exactly as they would if no lightsourcing were applied (that is, as they've appeared in the last two chapters) and 0 indicating that all pixels have the same color: black. Values between 0 and 32 will indicate degrees of brightness. For instance, an intensity value of 16 will indicate that the pixels on the surface appear half as bright as they do at full intensity.

To perform lightsourcing on a pixel, we must calculate the distance to the pixel, apply the lightsourcing formula to get the value of *intensity_ratio*, and multiply *intensity_ratio* by the brightness of the pixel to determine the actual color that we are to place on the screen.

But how do we know what the brightness of the pixel is? When we copy a pixel from the texture-map buffer to the screen, all we know is the palette number of the pixel. We don't even know what color it is (which can vary from palette to palette) much less how bright it is.

You'll recall that the VGA color registers contain three numbers in the range 0 to 63 for every color in the palette. These numbers represent the red, green, and blue intensities of the color. For instance, a medium gray shade might be represented by the numbers 32,32,32, meaning that red, green, and blue are all at about half their maximum intensities. A bright red color might be represented by the numbers 63,0,0, indicating that red is at full intensity while green and blue are at minimum intensity.

To change the brightness of the red, green, or blue component of a color, we must increase or decrease the number representing that component. If we increase the number representing the green component of a color, for instance, we make that component brighter. If we decrease the number, we make that component darker. However, if we raise and lower these numbers indiscriminately, we can actually change the color. Increasing the green component of a color may make the green component brighter, but it also makes the color itself seem more green.

To change the brightness of a color without changing its hue, we must be careful to increase or decrease all three color components together. Furthermore, we must take care to preserve the relationship *between* these components, because it is that relationship that establishes the hue. For instance, all colors in which the three color components are equal are shades of gray. We can increase or decrease the intensity of the color by increasing or decreasing the color components. But if we want the color to stay gray, we must be sure the three components are always equal.

This isn't hard to do. We just have to multiply all three components by the same lightsourcing factor. The factor that we'll be using is the *intensity_ratio* component from the formula above. You'll recall that this number is a fraction in the range 0 to 1. To find the lightsourced RGB components of a pixel, we need merely multiply this fraction by the non-lightsourced RGB values for the pixel. Suppose, for instance, that we wish to place a pixel on the display that has non-lightsourced RGB components of 48,32,16. And suppose that, after solving the lightsourcing equation for the distance between the viewer and the pixel, we determine that *intensity_ratio* has a value of 0.5 (meaning that the pixel should appear half as bright as in full light). To determine the lightsourced value of the pixel, we simply multiply each of the RGB components by 0.5, producing lightsourced RGB values of 24,16,8. What could be easier?

A lot of things, actually. While all of this sounds fairly simple in theory, the way that colors are handled by the VGA adapter makes lightsourcing a great deal more difficult than it should be. For one thing, it's rare that multiplying RGB

values by *intensity_ratio* will produce integral values. More likely, it will produce three fractional values, such as 17.9,50.231,39.8. This is a minor problem, since it's fairly easy to round such numbers to the nearest integer, and the difference will generally be unnoticeable to the viewer. Far more troublesome is the fact that, despite being able to produce 64 x 64 x 64 (or 262,144) different colors, the VGA adapter in mode 13h only allows us to place 256 of them on the screen at one time. So just knowing what color needs to be placed on the display to represent the lightsourced value of a pixel isn't enough. We must then find out if that color is present in the 256-color screen palette. If it isn't (and it usually won't be), we need to find the closest approximation of that color and use it instead.

This can be quite a time-consuming process. Performing these operations while casting rays through the maze could slow our ray-casting engine to a crawl. Fortunately, there's no need to perform these calculations on the fly. We can perform them in advance, save the results in a couple of arrays, and pass those arrays to the ray-casting engine, which can simply look up relevant values as it requires them.

The first of these arrays will be a two-dimensional array — that is, an array of arrays — in which there are 32 entries for every color in the 256-color palette. This may sound like rather a large array, but it will only take up 8 kilobytes of our precious computer memory. The advantages we gain from this array will be well worth the sacrifice. The 32 entries for each palette color will contain the palette numbers of other palette colors that come closest to approximating the way that that color would look at each of the 32 light levels. (Actually, there are *33* light levels, counting 0, so this array will really contain 33 entries for each of the 256 palette colors.)

This is a potentially confusing concept, so I'll pause here for a moment to explain it in more detail. Suppose that palette color 10 is a shade of gray with RGB values of 32,32,32. Since we're using an intensity scale that runs from 0 to 32, it's not difficult to see what the ideal list of lightsourced colors for this color would look like. At an intensity level of 0, it would have RGB values of 0,0,0. (Actually, every color has RGB values of 0,0,0 — pure black at intensity 0,0,0.) At intensity 1, it would have RGB values of 1,1,1. At intensity 2, it would have RGB values of 2,2,2. And so on, all the way up to intensity 32, where it would have RGB values of 32,32,32.

Entry 10 in the lightsourcing array would consist of the numbers of the 32 colors that represent lightsourced versions of color 10. Ideally, the first entry in the list (corresponding to a light intensity of 0) would be the number of the color with RGB values of 0,0,0. Since that color is pure black, it almost certainly *will* be in the list, since almost every palette contains black. If black is color 0 in the palette, the first entry in the array for color 10 will be 0. If the color with RGB

values of 1,1,1 is also in the palette, its number will be entry 2 in the list. If it isn't, then entry 2 will contain the number of the color that most closely approximates this color. For instance, if the color with RGB values 2,1,1 is in the palette, its number may appear in entry 2 instead. It's not quite the right color, but it's close enough. Entry 3 will contain the color that comes closest to having RGB values of 2,2,2, entry 4 will contain the color that comes closest to having RGB values of 3,3,3, and so forth. The last entry will represent the palette color that best represents the color at intensity 32. Since this is full brightness, the last entry will always point to the color itself. For color 10, the last entry will be 10.

Once we've created these tables, finding the lightsourced color of a pixel will be a simple matter of calculating an intensity number in the range 0 to 32 that represents the intensity of the light falling on that surface and looking up the equivalent entry in the table of light source values for that color. We can calculate the intensity number by solving the lightsourcing formula for the value of *intensity_ratio,* then multiplying *intensity_ratio* by 32. All clear?

For example, if the lightsourcing tables are contained in the two-dimensional array *lightsource[][],* where the number in the first pair of brackets represents the palette number of the color and the number in the second pair of brackets represents the light intensity, we can find the lightsourced value of a pixel like this:

```
float intensity_ratio = intensity_of_source / distance * MULTIPLIER;
if (intensity_ratio) > 1.0) intensity_ratio = 1.0;
int intensity = intensity_ratio*32;
int lightsourced_color = lightsource[color][intensity_ratio*32];
```

where *color* is the palette color of the non-lightsourced pixel, *distance* is the distance to the pixel, and *lightsourced_color* is the palette color that will actually be placed on the video display.

You'll notice that we truncate the value of *intensity_ratio*, so that it can never become larger than 1.0 and *increase* the maximum light intensity value. Surprisingly, *intensity_ratio* not only *can* go over 1.0, but it frequently *will*, depending on the values given to *light_intensity* and MULTIPLIER. Setting MULTIPLIER to a value greater than 1 (or *light_intensity* to a value greater than 32) has the effect of producing a circle of full brightness around the player character, which not only looks realistic but will save the player considerable eyestrain.

Before we can create a lightsourced version of the *draw_maze()* function, we'll need to create a lightsourcing table and save it in a two-dimensional array. This is a task that can take up a lot of CPU time, since it involves searching the entire palette 32 times for each of the 256 palette colors to find lightsourced equivalents

for those colors. So we'll write a special utility program that will create the table and save it to a disk file. Our ray-casting program can then read the data during its initialization.

We'll call this utility program MAKELITE.CPP. It will use the *loadPCX()* function to read a PCX file from the disk and will create lightsourcing tables for that palette. Thus, if we are going to use the tiles and palette in the file IMAGES.PCX in our ray-casting program, and therefore want to base the lightsourcing tables on that file, we can invoke MAKELITE by typing:

```
MAKELITE IMAGES.PCX
```

at the DOS prompt. This means that we'll need to include code at the beginning of the *main()* function of the program to make sure that there's a file name from the command line:

```
if (argc<2) {
  printf("You must type a filename.\n");
  exit(1);
}
```

Remember that there will always be at least one parameter on the command line: the program's file name. Thus we check to see if there are *2* values, the second presumably being the PCX file name.

We'll also allow the user to pass three optional numeric parameters on the command line, after the file name. These parameters will represent the target values for the red, green, and blue components of the palette colors. If no parameters are typed on the command line, these values will default to 0,0,0, meaning that we want the palette colors in our lightsource tables to converge on the color black as the distance from the viewer increases. However, there will be times when we want these values to converge on other colors, such as white or blue, for certain special effects. I'll talk more about this in a moment.

We'll first initialize these values to black:

```
int redtarget=0;
int greentarget=0;
int bluetarget=0;
```

then we'll check to see if alternate values have been typed on the command line. If so, we'll use them:

```
if (argc>=3) redtarget=atof(argv[2]);
if (argc>=4) bluetarget=atof(argv[3]);
if (argc>=5) greentarget=atof(argv[4]);
```

The process that follows will be fairly time-consuming, especially on slower machines. (It takes about 15 seconds on my 40-mHz 486.) So that the user won't think the computer has locked up, we'll print periodic messages on the display

explaining what the program is doing. Since the first thing we'll do is load the PCX file and extract its palette, that's what we'll indicate in the first message:

```
printf("Loading palette.\n");
```

We'll link our well-worn PCX routines to the program, so all we need to do to load the file is to call the *loadPCX()* function using the command line parameter *argv[1]* as the file name:

```
if (loadPCX(argv[1],&litepal)) exit(1);
```

This loads the file into *litepal*, which will need to be declared earlier in the program as a *PCX_struct*. (You'll recall that the *PCX_struct* type is defined in the file PCX.H, which we'll need to include.) As usual, we check for a valid load, exiting the program if *loadPCX* tells us that an error has occurred.

For purposes of clarity, we'll copy the palette from the *palette* field of the *litepal* structure into three arrays of integer variables, which we'll call *red[], green[],* and *blue[].* Each of these arrays will have 256 entries, one for each palette color. The individual entries will contain the red, green, and blue values for the equivalent palette color. Strictly speaking, this step is unnecessary, since we already have access to these values in the *litepal* structure. But the use of the three arrays will make the code that follows clearer. And this is hardly a time-critical program, which will suffer from the extra CPU cycles. We'll do the copying with a *for()* loop:

```
for (int color=0; color<PALETTESIZE; color++) {
  red[color]=litepal.palette[color*3];
  green[color]=litepal.palette[color*3+1];
  blue[color]=litepal.palette[color*3+2];
}
```

The constant PALETTESIZE will be set equal to 256, representing the number of colors in the mode 13h palette. This is also done for purposes of clarity, since it's unlikely that this program will ever be rewritten to handle palettes of a different size.

Now it's time to calculate the palette tables. We'll print a message to the user so that they'll know what the program is up to:

```
printf("Calculating palette tables");
```

We'll perform the palette calculations within a set of three nested *for()* loops. The outer loop will iterate through all 256 colors in the palette. The middle loop will iterate through all 33 intensity levels of that color. And the innermost loop will once again iterate through all 256 palette colors, searching for the closest match for the current color at the current light level. Here's the beginning of the outer loop:

```
for (color=0; color<PALETTESIZE; color++) {
```

Since the inner loop will repeat 256 times every time that the middle loop executes and the middle loop will execute 32 times for every time the outer loop executes (and the outer loop will execute 256 times altogether), the innermost instructions (which will contain several time-consuming floating point operations) will be repeated 256 x 33 x 256, or 2,162,688, times. This is where the user is most likely to decide that the program has crashed, so we'll need to print something on the screen while all of this is going on. We don't need to print *much* — a simple period (.) will do — and we don't need to print it often, so we'll just slip this instruction in at the head of the outer loop:

```
printf(".");
```

Since this instruction only repeats 256 times, it won't put much drag on the execution of the program. But it will print periods on the display often enough to let the user know that the computer hasn't given up the ghost.

Now we can begin the second loop. This loop will execute 33 times, once for every light intensity level:

```
for (int level=0; level<=MAXLIGHT; level++) {
```

MAXLIGHT is a constant that we'll set equal to the number of light levels, which for this program will be 32. Should you decide that you want to experiment with a larger or smaller number of light levels, you can redefine this constant and recompile the program. Or you can change MAXLIGHT to a variable and have the program read its value as a command line value.

Before we can start searching the palette for the closest equivalent to the current color at the current intensity level, we need to calculate the ideal RGB values for the lightsourced value of the color. How do we do this? As the middle loop repeats 33 times, once for every intensity level, we want to move the RGB values of this color in the direction of the target RGB values, which will usually be 0,0,0. Since the intensity values cover a range of 32 increments, we'll want to increment these values by 1/32nd of the distance between the current RGB values and the target values on each iteration of the loop. We can easily find out what 1/32nd of the distance would be by subtracting the target values from the current values (which we earlier moved to the *red[]*, *green[]*, and *blue[]* arrays) and dividing the differences by 32. (Actually, we'll divide it by MAXLIGHT, the constant we mentioned earlier.) Then we can multiply the differences by the current intensity level and add the value of the target color to the result to find the ideal RGB values:

```
float redlite=(float)(red[color]-redtarget) / MAXLIGHT * level +
redtarget;
float greenlite=(float)(green[color]-greentarget) / MAXLIGHT * level +
greentarget;
```

```
float bluelite=(float)(blue[color]-bluetarget) / MAXLIGHT * level +
bluetarget;
```

The ideal lightsourced RGB values for the current palette color at the current intensity are now stored in the floating point variables *redlite, greenlite,* and *bluelite*. Most of the time, there will be no perfect fit for these intensities in the 256-color palette. (As I mentioned earlier, these values may not even be whole numbers.) So we have to search the palette to find the closest fit. We'll do this by assigning a proximity score to each palette color, with lower scores indicating closer matches than higher scores. The color with the lowest score will be assigned as the lightsourced value for the current color at the current intensity. To determine which score is lowest, we'll maintain a floating point variable called *bestscore,* which will hold the lowest score of all the palette colors so far searched. If a lower palette color is found, *bestscore* will be assigned that value instead. We'll initialize *bestscore* by setting it to the highest possible floating point value, higher than any possible score that will be assigned to a palette color, so that it will immediately be replaced by the score of the first color in the palette:

```
float bestscore=MAXFLOAT;
```

MAXFLOAT is a constant defined in the header file VALUES.H that represents the largest possible floating point number that the compiler can handle.

To perform the search, we'll iterate (again) through all of the palette colors:

```
for (int color2=0; color2<PALETTESIZE; color2++) {
```

Calculating the proximity score for a palette color isn't hard. We can find the proximity scores for the individual RGB values of a color simply by subtracting those RGB values from the ideal RGB values that we stored in *redlite, greenlite,* and *bluelite:*

```
redscore=fabs(red[color2]-redlite);
greenscore=fabs(green[color2]-greenlite);
bluescore=fabs(blue[color2]-bluelite);
```

Notice that we use the *fabs()* function (defined in the standard header file MATH.H) to get the absolute value of the differences, since only the distance between the values matters, not the direction of that distance. We don't care if the RGB components are brighter or darker than the ideal values, only that the difference is relatively small. To calculate an overall proximity score for this color, we add the proximity scores for the RGB values:

```
float score=redscore+greenscore+bluescore;
```

Is this score better (that is, smaller) than the previous best score?

```
if (score<bestscore) {
```

If it is, we replace the previous best score:

```
bestscore=score;
```

It's not enough simply to save the best score, of course. We also need to remember which palette color received that score. We'll be placing the lightsourcing data in a two-dimensional array called *litesource[][]*, which will be defined at the beginning of the program. So we might as well go ahead and place the number of the top-scoring palette color in the entry of that array corresponding to the current level and color. If another palette color gets a better score, the number of that color will replace this one:

```
            litesource[level][color]=color2;
          }
        }
      }
    }
```

That not only ends the *if()* statement but all three of the nested loops. When we reach this point in the program, the lightsourcing array will have been completed. So let's write it to the disk. We'll let the user know what's happening:

```
printf("\nWriting LITESORC.DAT.\n");
```

Then we'll open a file called LITESORC.DAT to stuff the data into:

```
if ((handle=fopen("litesorc.dat","wb"))==NULL) {
  perror("Error");
  exit;
}
```

This will terminate the program if an error occurs. The *perror()* function, the prototype for which is in the standard header file FCNTL.H, will print the word "Error" followed by a colon and a message explaining the nature of the error.

If all goes well up to this point, we can write the contents of the *litesource[][]* array to the file with a single instruction:

```
fwrite(litesource,MAXLIGHT+1,PALETTESIZE,handle);
```

All that remains is to close the file, let the user know that everything's hunky dory, and terminate the *main()* function:

```
  fclose(handle);
  printf("Done!\n");
}
```

The MAKELITE.CPP Program

The complete text of the MAKELITE.CPP program appears in Listing 11-1.

Listing 11-1 The MAKELITE.CPP program

```
//
// MAKELITE.CPP
//
// Utility for generating lightsourcing tables.
// Written by Christopher Lampton
// for Gardens of Imagination (Waite Group Press)
//

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <values.h>
#include <conio.h>
#include <io.h>
#include <fcntl.h>
#include "pcx.h"

const MAXLIGHT=32;
const PALETTESIZE=256;

pcx_struct litepal;
int red[PALETTESIZE],green[PALETTESIZE],
    blue[PALETTESIZE];
unsigned char litesource[MAXLIGHT+1][PALETTESIZE];
float redscore,greenscore,bluescore;
FILE *handle;

void main(int argc,char* argv[])
{

  // Set default target to black:
  int redtarget=0;
  int greentarget=0;
  int bluetarget=0;

  // Check for filename on the command line:
  if (argc<2) {
    printf("You must type a filename.\n");
    exit(1);
  }

  // Check for target values on command line:
  if (argc>=3) redtarget=atof(argv[2]);
  if (argc>=4) bluetarget=atof(argv[3]);
  if (argc>=5) greentarget=atof(argv[4]);
```

continued on next page

continued from previous page

```
  // Let user know what's going on:
  printf("Loading palette.\n");

  // Load the PCX file:
  if (loadPCX(argv[1],&litepal)) exit(1);

  // Copy palette to RGB arrays:
  for (int color=0; color<PALETTESIZE; color++) {
    red[color]=litepal.palette[color*3];
    green[color]=litepal.palette[color*3+1];
    blue[color]=litepal.palette[color*3+2];
  }

  // Let user know what we're about to do:
  printf("Calculating palette tables");

  // Iterate through all 256 palette colors:
  for (color=0; color<PALETTESIZE; color++) {

    // Let user know we haven't died:
    printf(".");

    // Iterate through all 32+1 intensity levels:
    for (int level=0; level<=MAXLIGHT; level++) {

      // Calculate ideal lightsourced RGB values:
      float redlite=(float)(red[color]-redtarget) / MAXLIGHT
              * level + redtarget;
      float greenlite=(float)(green[color]-greentarget) / MAXLIGHT
              * level + greentarget;
      float bluelite=(float)(blue[color]-bluetarget) / MAXLIGHT
              * level + bluetarget;

      // Initialize score to very large number:
      float bestscore=MAXFLOAT;

      // Search the palette for closest match:
      for (int color2=0; color2<PALETTESIZE; color2++) {

        // Assign proximity score to color:
        redscore=fabs(red[color2]-redlite);
        greenscore=fabs(green[color2]-greenlite);
        bluescore=fabs(blue[color2]-bluelite);
        float score=redscore+greenscore+bluescore;

        // Is this better (i.e. smaller) than previous best?
        if (score<bestscore) {

          // If so, replace best score:
          bestscore=score;

          // And remember which color got it:
```

```
            litesource[level][color]=color2;
          }
        }
      }
    }

    // Tell user that the worst is over:
    printf("\nWriting LITESORC.DAT.\n");

    // Try to open disk file:
    if ((handle=fopen("litesorc.dat","wb"))==NULL) {
      perror("Error");
      exit;
    }

    // If successful, write lightsource tables to file:
    fwrite(litesource,MAXLIGHT+1,PALETTESIZE,handle);

    // Close up shop:
    fclose(handle);

    // And tell the user to breathe easy:
    printf("Done!\n");
}
```

Using the Lightsourcing Tables

Now that we've got the light source data, what are we going to do with it? Earlier, I showed you the instructions necessary for calculating the color of a pixel based on its distance from the viewer using the lightsourcing equation and the light source tables. The light source tables save us from having to perform a time-consuming search of the palette before we write a pixel to the display, but even the relatively simple lightsourcing formula, which contains a division and a multiplication, is too time-consuming to perform while ray casting is in progress, since it must be performed for every pixel in the viewport. So, in order to further reduce the time required to perform lightsourcing on a pixel, we'll place the results of the lightsourcing formula in a second array, which we'll call *litelevel[]*. There will be one entry in this array for every possible distance that a pixel can be from the viewer, representing the intensity of the light (on a scale of 0 to 32) at that distance. In a maze with GRIDSIZE squares across and down, where each square is 64 by 64 pixels in size, the maximum possible distance will be GRIDSIZE x 64, which will cover the case in which the viewer is on one side of the maze looking toward the far side. In the 16 by 16 mazes we've used so far in this book, there would need to be 4,096 entries in this array. Since there are only 33 possible light intensities, we can use an *unsigned char* array, which will only take up 4 kilobytes of memory. Cheap at the price.

Calculating the values in the *litelevel[]* array will take a lot less CPU time than calculating the lightsourcing values for each palette color, but it's still sufficiently time-consuming that we'll want to get it out of the way during program initialization. For that reason, we won't put it in the *draw_maze()* function, but will require the calling program to calculate it and pass a pointer to *draw_maze()* as a parameter. So we're going to deviate this one time from the tradition that we established much earlier in this book of writing the *draw_maze()* function and then writing a small demo program to call it. Instead, we'll write the demo program first.

This demo program will be based on the RAYDEMO.CPP program from chapter 9. I'll only discuss the differences between this lightsourcing demo, which we'll call LITEDEMO.CPP, and the earlier demo. The major difference will be the calculation of the *litelevel()* array. The array will be declared like this:

```
unsigned char *litelevel;
```

Before we can calculate the values, we must allot space for the data:

```
litelevel=new unsigned char[MAXDISTANCE];
```

Earlier, we'll initialize the constant MAXDISTANCE to the maximum possible distance that a light ray can travel through the maze:

```
const MAXDISTANCE=64*GRIDSIZE;
```

The data will be calculated in a *for()* loop, which will iterate through all possible distances:

```
for (int distance=1; distance<MAXDISTANCE; distance++) {
```

Next, we apply the lightsourcing formula that I described a few pages ago:

```
  float intensity_ratio=(float)intensity/distance*MULTIPLIER;
  if (intensity_ratio>1.0) ratio=1.0;
  litelevel[distance]=ratio*MAXLIGHT;
}
```

We now have an array of light intensities for all 4,096 distances.

We'll also need to load the lightsourcing tables into the array from the disk file that MAKELITE.CPP stored them in. We'll declare the array like this:

```
unsigned char litetable[MAXLIGHT+1][PALETTESIZE];
```

The instructions for loading the array are almost identical to those that we used in MAKELITE.CPP to save it:

```
if ((handle=fopen("litesorc.dat","rb"))==NULL) {
  perror("Error");
  exit;
}
```

```
fread(litetable,MAXLIGHT+1,PALETTESIZE,handle);
fclose(handle);
```

Up until now, we've given the user the option of placing a floating point number on the command line to represent the desired viewing angle. This time, let's give the user a couple of additional options:

```
if (argc>=2) viewing_angle=atof(argv[1]);
if (argc>=3) intensity=atof(argv[2]);
if (argc>=4) ambient_level=atof(argv[3]);
```

The first line checks for the viewing angle parameter and reads it off the command line if it's found. The second line loads a value for the intensity of the light source (which we'll set to 32 — full brightness — by default). The third line reads the ambient lighting level. We mentioned ambient lighting in the chapter on ray tracing. Ambient light is light that appears to come from all directions at once, filling shadows with a certain minimum light level. We'll add the ambient light level to all pixels on top of the light received from the player's "torch," but will not let the overall light level exceed 32. An ambient level of 0 will make the room appear to be lit only by the torch. A level of 32 will make all surfaces appear the way they would in full light (that is, the way they did in the ray-casting chapter). You can tweak the ambient value to levels in between these two to find the level you like best. Even in the darkest dungeon a small ambient level is desirable; otherwise the screen will turn pitch black if the player's torch should go out. Of course, that may be the effect that you're looking for.

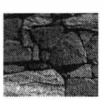
Finally, the call to our lightsourced version of the *draw_maze()* function will look like this:

```
draw_maze(walls,flor,ceiling,screen,xview,yview,viewing_angle,
          viewer_height,ambient_level,textmaps.image,
          litetable,litelevel);
```

This is similar to the way in which the ray-cast version in chapter 9 was called, with the familiar pointers to the *walls[], flor[],* and *ceiling[]* arrays, but we've added pointers to the *litetable* and *litelevel* arrays.

The LITEDEMO.CPP Program

The complete text of the LITEDEMO.CPP program appears in Listing 11-2.



Listing 11-2 The LITEDEMO.CPP program

```
// LITEDEMO.CPP
//
```

continued on next page

continued from previous page

```
// Calls ray-casting function to draw lightsourced
// view of maze.
//
// Written by Christopher Lampton for
// Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "screen.h"
#include  "pcx.h"
#include  "litesorc.h"

const GRIDSIZE=16;
const MAXDISTANCE=64*GRIDSIZE;
const NUM_IMAGES=15;
const float MULTIPLIER=3;

pcx_struct textmaps;
FILE *handle;
unsigned char litetable[MAXLIGHT+1][PALETTESIZE];
unsigned char *litelevel;

map_type walls={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 7, 5, 0, 5, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 5},
 { 7, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 5},
 { 7, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 5},
 { 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 4, 0, 0, 0, 5},
 {12, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,11, 0, 0, 0, 5},
 {11, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,10, 0, 0, 5},
 {11, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,10, 0, 0, 5},
 {11, 7, 7, 0, 0, 0, 0, 0, 0, 0, 0,11,10, 0, 0, 5},
 {12, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,11, 0, 0, 0, 5},
 {12, 7, 7, 0, 0, 0, 5, 1, 0, 1, 0, 0, 0, 0, 0, 5},
 { 7, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 5},
 { 7, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,12},
 { 7, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,11},
 { 7, 6, 6, 6, 6, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,10},
 { 7, 5, 5, 5, 5, 5, 5,15,15,15, 5, 5, 5, 5, 5, 5}
};

map_type flor={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 8, 8, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 8, 8, 8, 8, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 8, 8, 8, 5, 5, 5, 5, 5, 8, 5, 5, 5, 5},
```

```
 { 5, 5, 5, 5, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
};

map_type ceiling={
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,12,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13}
};

float viewing_angle=3.14;
float intensity=MAXLIGHT;
int viewer_height=32;
int ambient_level=0;
int xview=8*64+32;
int yview=8*64+32;

void main(int argc,char* argv[])
{
  // Read arguments from command line if present:
  if (argc>=2) viewing_angle=atof(argv[1]);
  if (argc>=3) intensity=atof(argv[2]);
  if (argc>=4) ambient_level=atof(argv[3]);

  // Load texture map images:
  if (loadPCX("images.pcx",&textmaps)) exit(1);

  // Load lightsourcing tables:
  if ((handle=fopen("litesorc.dat","rb"))==NULL) {
    perror("Error");
    exit;
  }
  fread(litetable,MAXLIGHT+1,PALETTESIZE,handle);
  fclose(handle);
```

continued on next page

continued from previous page

```
  // Initialize array of light levels:
  litelevel=new unsigned char[MAXDISTANCE];

  // Calculate light intensities for all possible
  // distances:
  for (int distance=1; distance<MAXDISTANCE; distance++) {
    float ratio=(float)intensity/distance*MULTIPLIER;
    if (ratio>1.0) ratio=1.0;
    litelevel[distance]=ratio*MAXLIGHT;
    int dummy=0;
  }

  // Point variable at video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Set mode 13h:
  setmode(0x13);

  // Set the palette:
  setpalette(textmaps.palette);

  // Clear the screen:
  cls(screen);

  // Draw a ray-cast view of the maze
  draw_maze(walls,flor,ceiling,screen,xview,yview,viewing_angle,
              viewer_height,ambient_level,textmaps.image,
              litetable,litelevel);

  // Wait for user to hit a key:
  while (!kbhit());

  // Release memory
  delete textmaps.image;
  delete litelevel;

  // Reset video mode and exit:
  setmode(oldmode);
}
```

Lightsourcing in Three Lines or Less

The prototype for the function, which will go in the file LITESORC.H, looks like this:

```
void draw_maze(map_type map,map_type floor,map_type ceiling,
                      char far *screen,int xview,int yview,
                      float viewing_angle,int viewer_height,
                      int ambient_level,unsigned char far *textmaps,
```

```
unsigned char[MAXLIGHT+1][PALETTESIZE],
unsigned char *litelevel);
```

The function itself is in the file LITESORC.CPP. It is nearly identical to the ray-cast version of the *draw_maze()* function in chapter 9. All that has changed, other than the prototypes, are the lines that actually place data on the display. Instead of simply copying the pixel data from the texture-map buffer to video RAM, we'll do something slightly more complicated. We'll use the *distance* variable (which contains the distance, in floor pixel units, from the viewer to the pixel being drawn) as an index into the *litelevel* array to find the light level for that distance, then add the ambient term to that value:

```
int level=litelevel[distance]+ambient_level;
```

Since the ambient term may take this value over 32, we'll chop it off in that level:

```
if (level>MAXLIGHT) level=MAXLIGHT;
```

The constant MAXLIGHT is assigned a value of 32 in the LITESORC.H file, should we be overcome with an urge to change the number of light levels.

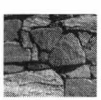
Now that we have the light level, we can use it as an index into the *litesource[][]* array to find the lightsourced color for the current pixel. We'll use the non-lightsourced color for the pixel, which is at offset *tileptr* in the *textmaps[]* array as the second index for this two-dimensional array. Then we'll copy the resulting value into video RAM:

```
screen[offset]=litesource[level][textmaps[tileptr]];
```

The *draw_maze()* function writes pixels to the display in three different places, once when drawing the walls, once when drawing the floors, and once when drawing the ceiling. We'll use these same three lines of code in all three places. And those, believe it or not, are all the changes that are necessary to add lightsourcing to the core of our ray-casting engine. By placing all of the tough work in the calling module and in the MAKELITE program, we've actually left very little extra work for *draw_maze()* to do.

The Lightsourced draw_maze() Function

The lightsourced version of the *draw_maze()* function appears in Listing 11-3.



Listing 11-3 The lightsourced draw_maze() function

```
// LITESORC.CPP
//
```

continued on next page

continued from previous page

```
// Function to draw lightsourced walls, floors and
// ceilings using ray casting.
// Written by Christopher Lampton for
// GARDENS OF IMAGINATION (Waite Group Press).
//

#include <stdio.h>
#include <math.h>
#include "litesorc.h"
#include "pcx.h"

// Constant definitions:
const WALL_HEIGHT=64;           // Height of wall in pixels
const VIEWER_DISTANCE=192;      // Viewer distance from screen
const VIEWPORT_LEFT=0;          // Dimensions of viewport
const VIEWPORT_RIGHT=319;
const VIEWPORT_TOP=0;
const VIEWPORT_BOT=199;
const VIEWPORT_HEIGHT=VIEWPORT_BOT-VIEWPORT_TOP;
const VIEWPORT_CENTER=VIEWPORT_TOP+VIEWPORT_HEIGHT/2;
const GRIDSIZE=16;
const MAXDISTANCE=64*GRIDSIZE;

void draw_maze(map_type map,map_type floor,map_type ceiling,
                char far *screen,int xview,int yview,
                float viewing_angle,int viewer_height,
                int ambient_level,unsigned char far * textmaps,
                unsigned char litesource[MAXLIGHT+1][PALETTESIZE],
                unsigned char *litelevel)

// Draws a ray-cast image in the viewport of the maze represented
// in array MAP[], as seen from position XVIEW, YVIEW by a
// viewer looking at angle VIEWING_ANGLE where angle 0 is due
// north. (Angles are measured in radians.)

{
// Variable declarations:
int sy,offset;                  // Pixel y position and offset
float xd,yd;                    // Distance to next wall in x and y
int grid_x,grid_y;              // Coordinates of x and y grid lines
float xcross_x,xcross_y;        // Ray intersection coordinates
float ycross_x,ycross_y;
unsigned int xdist,ydist;       // Distance to x and y grid lines
int xmaze,ymaze;                // Map location of ray collision
int distance;                   // Distance to wall along ray
int tmcolumn;                   // Column in texture map
float yratio;

// Loop through all columns of pixels in viewport:
for (int column=VIEWPORT_LEFT; column<VIEWPORT_RIGHT; column++) {

    // Calculate horizontal angle of ray relative to
```

```
//  center ray:
float column_angle=atan((float)(column-160)
                    / VIEWER_DISTANCE);

// Calculate angle of ray relative to maze coordinates
float radians=viewing_angle+column_angle;

// Rotate endpoint of ray to viewing angle:
int x2 = -1024 * (sin(radians));
int y2 = 1024 * (cos(radians));

// Translate relative to viewer's position:
x2+=xview;
y2+=yview;

// Initialize ray at viewer's position:
float x=xview;
float y=yview;

// Find difference in x,y coordinates along ray:
int xdiff=x2-xview;
int ydiff=y2-yview;

// Cheat to avoid divide-by-zero error:
if (xdiff==0) xdiff=1;

// Get slope of ray:
float slope = (float)ydiff/xdiff;

// Cheat (again) to avoid divide-by-zero error:
if (slope==0.0) slope=.0001;

// Cast ray from grid line to grid line:
for (;;) {

  // If ray direction positive in x, get next x grid line:
  if (xdiff>0) grid_x=((int)x & 0xffc0)+64;

  // If ray direction negative in x, get last x grid line:
  else grid_x=((int)x & 0xffc0) - 1;

  // If ray direction positive in y, get next y grid line:
  if (ydiff>0) grid_y=((int)y & 0xffc0) +64;

  // If ray direction negative in y, get last y grid line:
  else grid_y=((int)y & 0xffc0) - 1;

  // Get x,y coordinates where ray crosses x grid line:
  xcross_x=grid_x;
  xcross_y=y+slope*(grid_x-x);

  // Get x,y coordinates where ray crosses y grid line:
```

continued on next page

continued from previous page

```
      ycross_x=x+(grid_y-y)/slope;
      ycross_y=grid_y;

      // Get distance to x grid line:
      xd=xcross_x-x;
      yd=xcross_y-y;
      xdist=sqrt(xd*xd+yd*yd);

      // Get distance to y grid line:
      xd=ycross_x-x;
      yd=ycross_y-y;
      ydist=sqrt(xd*xd+yd*yd);

      // If x grid line is closer...
      if (xdist<ydist) {

        // Calculate maze grid coordinates of square:
        xmaze=xcross_x/64;
        ymaze=xcross_y/64;

        // Set x and y to point of ray intersection:
        x=xcross_x;
        y=xcross_y;

        // Find relevant column of texture map:
        tmcolumn = (int)y & 0x3f;

        // Is there a maze cube here? If so, stop looping
        if (map[xmaze][ymaze]) break;
      }
      else { // If y grid line is closer:

        // Calculate maze grid coordinates of square:
        xmaze=ycross_x/64;
        ymaze=ycross_y/64;

        // Set x and y to point of ray intersection:
        x=ycross_x;
        y=ycross_y;

        // Find relevant column of texture map:
        tmcolumn = (int)x & 0x3f;

        // Is there a maze cube here? If so, stop looping:
        if (map[xmaze][ymaze]) break;
      }
    }

    // Get distance from viewer to intersection point:
    xd=x-xview;
    yd=y-yview;
    distance=(long)sqrt(xd*xd+yd*yd)*cos(column_angle);
```

```
if (distance==0) distance=1;

// Calculate visible height of wall:
int height = VIEWER_DISTANCE * WALL_HEIGHT / distance;

// Calculate bottom of wall on screen:
int bot = VIEWER_DISTANCE * viewer_height
          / distance + VIEWPORT_CENTER;

// Calculate top of wall on screen:
int top = bot - height + 1;

// Initialize temporary offset into texture map:
      int t=tmcolumn;

// If top of current vertical line is outside of
// viewport, clip it:

int dheight=height;
int iheight=IMAGE_HEIGHT;
yratio=(float)WALL_HEIGHT/height;
if (top < VIEWPORT_TOP) {
  dheight-=(VIEWPORT_TOP - top);
  t+=(int)((VIEWPORT_TOP-top)*yratio)*320;
  iheight -= ((VIEWPORT_TOP-top)*yratio);
  top=VIEWPORT_TOP;
}
if (bot > VIEWPORT_BOT) {
  dheight -= (bot - VIEWPORT_BOT);
  iheight -= (bot - VIEWPORT_BOT)*yratio;
  bot=VIEWPORT_BOT;
}

// Point to video memory offset for top of line:
offset = top * 320 + column;

// Initialize vertical error term for texture map:
int tyerror=64;

// Which graphics tile are we using?
int tile=map[xmaze][ymaze]-1;

// Find offset of tile and column in bitmap:
unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;

// Loop through the pixels in the current vertical
// line, advancing OFFSET to the next row of pixels
// after each pixel is drawn.
for (int h=0; h<iheight; h++) {

  // Are we ready to draw a pixel?
```

continued on next page

continued from previous page

```
      while (tyerror>=IMAGE_HEIGHT) {

        // If so, draw it:
        int level=litelevel[distance]+ambient_level;
        if (level>MAXLIGHT) level=MAXLIGHT;
        screen[offset]=litesource[level][textmaps[tileptr]];

        // Reset error term:
        tyerror-=IMAGE_HEIGHT;

        // And advance OFFSET to next screen line:
        offset+=320;
      }

      // Incremental division:
      tyerror+=height;

      // Advance TILEPTR to next line of bitmap:
      tileptr+=320;
    }

    // Step through floor pixels:
    for (int row=bot+1; row<=VIEWPORT_BOT; row++) {

      // Get ratio of viewer's height to pixel height:
      float ratio=(float)viewer_height/(row-100);

      // Get distance to visible pixel:
      distance=ratio*VIEWER_DISTANCE/cos(column_angle);

      // Rotate distance to ray angle:
      int x = - distance * (sin(radians));
      int y = distance * (cos(radians));

      // Translate relative to viewer coordinates:
      x+=xview;
      y+=yview;

      // Get maze square intersected by ray:
      int xmaze = x / 64;
      int ymaze = y / 64;

      // Find relevant column of texture map:
      int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);

      // Which graphics tile are we using?
      int tile=floor[xmaze][ymaze];

      // Find offset of tile and column in bitmap:
      unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
        *IMAGE_WIDTH+t;
```

```
      // Calculate video offset of floor pixel:
      offset=row*320+column;

      // Draw pixel:
      int level=litelevel[distance]+ambient_level;
      if (level>MAXLIGHT) level=MAXLIGHT;
      screen[offset]=litesource[level][textmaps[tileptr]];
    }

    // Step through ceiling pixels:
    for (row=top-1; row>=VIEWPORT_TOP; --row) {

      // Get ratio of viewer's height to pixel height:
      float ratio=(float)(WALL_HEIGHT-viewer_height)/(100-row);

      // Get distance to visible pixel:
      distance=ratio*VIEWER_DISTANCE/cos(column_angle);

      // Rotate distance to ray angle:
      int x = - distance * (sin(radians));
      int y = distance * (cos(radians));

      // Translate relative to viewer coordinates:
      x+=xview;
      y+=yview;

      // Get maze square intersected by ray:
      int xmaze = x / 64;
      int ymaze = y / 64;

      // Find relevant column of texture map:
      int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);

      // Which graphics tile are we using?
      int tile=ceiling[xmaze][ymaze];

      // Find offset of tile and column in bitmap:
      unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
        *IMAGE_WIDTH+t;

      // Calculate video offset of floor pixel:
      offset=row*320+column;

      // Draw pixel:
      int level=litelevel[distance]+ambient_level;
      if (level>MAXLIGHT) level=MAXLIGHT;
      screen[offset]=litesource[level][textmaps[tileptr]];
    }

  }
}
```

To run the program, go to the LIGHT directory and type LITEDEMO. You'll see the familiar view of the ray-cast maze from chapter 9, but this time much of it will be shrouded in gloomy shadows (see Figure 11-5). Try fooling around with the *intensity* and *ambient_level* parameters to see what effect they have. (Remember that the first parameter on the command line is the angle, followed by the intensity and ambient parameters, in that order.) Type LITEDEMO 0 0 to see the maze in no light at all. Whoops, the screen is completely black! Not much reason to include a screen shot of that one. And LITEDEMO 0 32, where the ambient is turned up all the way, looks exactly like the screen shots at the end of chapter 9. But LITEDEMO 16 0 shows a spookier and darker dungeon (refer to Figure 11-6). And LITEDEMO 4 shows a dungeon made up of little besides shades of dark gray (Figure 11-7).

A little bit of ambient light can go a long way. Try LITEDEMO 32 4 to see the maze with a bright torch and some dim ambient lighting. But LITEDEMO 32 16, which raises the ambient level halfway to full brightness, doesn't look much different from the maze in full light.

Alternate Lightsourcing Effects

You don't have to use lightsourcing in your programs. Many popular maze games, such as Wolfenstein 3D, use no lightsourcing effects at all and still manage to

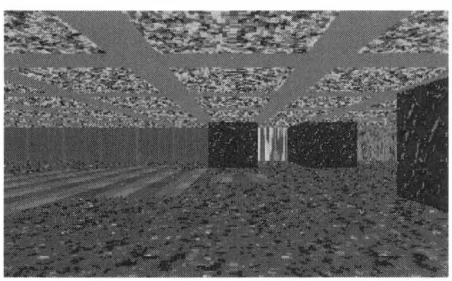


Figure 11-5 A gloomy maze produced by LITEDEMO.EXE

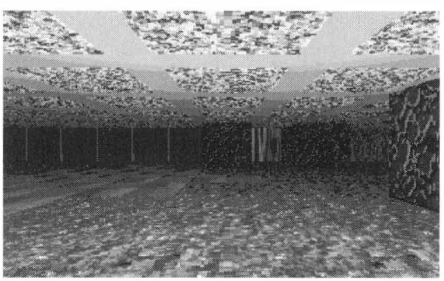


Figure 11-6 Turning down the intensity

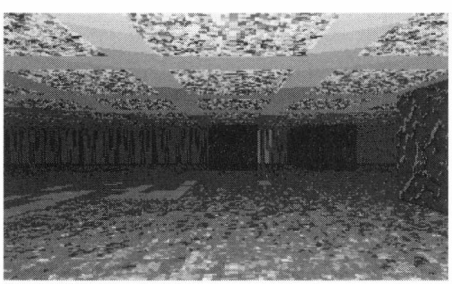


Figure 11-7 Almost in total darkness

enthrall players. But it's a nice special effect to use when you want to add something a little extra to your code. The best thing about using the MAKELITE program to generate lightsourcing tables is that you can use it with almost any palette whatsoever, not just with special palettes designed to work with lightsourced graphics. Of course, some palettes work better with this lightsourcing method than others. Since all the palette colors converge on black and dark gray at low light levels, it helps if there are several dark gray and brown colors in the palette. Otherwise, severe banding can occur in very low light, all colors being converted to one or two very dark colors by the lightsourcing routines.

Lightsourcing can be used with some special effects that you might not have thought had anything to do with light levels. The game Shadowcaster, from Origin Systems, uses lightsourcing to pull off some unexpected effects, such as scenes filled with fog (see Figure 11-8) and and underwater scenes (Figure 11-9). It's not hard to figure out that these must have been done with lightsourcing tables similar to the ones we've learned to create in this chapter. However, where the lightsourcing tables that we created for our demo program are based on a target color of pure black (with RGB values of 0,0,0), these fog and underwater effects must be based on lightsourcing tables with target values of white (63,63,63) and blue (0,0,63).

You can easily create a fog table with the MAKELITE utility, by going to the LIGHT directory and typing:

```
MAKELITE 63 63 63
```

Once the table is complete, run LITEDEMO again. This time the maze will be enshrouded with mist, as in Figure 11-10. To try and duplicate the underwater effect, type:

```
MAKELITE 0 0 63
```

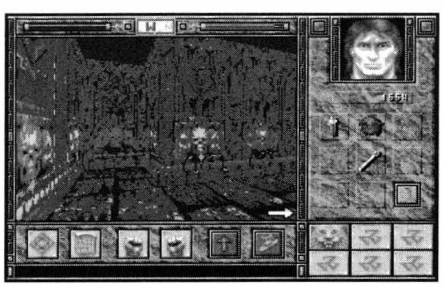


Figure 11-8 A scene from Origin's Shadowcaster, demonstrating how lightsourcing can create fog effects

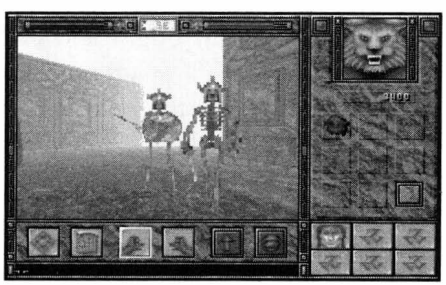


Figure 11-9 Another scene from Shadowcaster, with underwater lighting effects

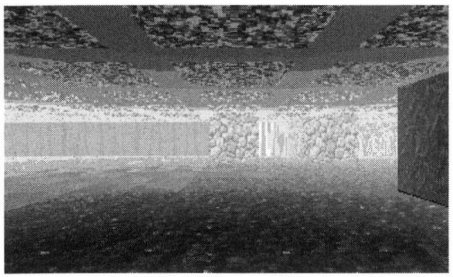


Figure 11-10 A mist-enshrouded maze created by using lightsource target values of 63,63,63

This effect is a bit less successful. Rather than appearing to be underwater, the maze appears to be enshrouded in *blue* fog this time. Typing LITEDEMO 8 0, which decreases the circle of light around the player, helps a bit, but too much detail is lost in the distance. To get a better underwater effect, you might take a tip from Shadowcaster and design a set of underwater texture-map tiles in which the color blue already predominates, with swirling lines that suggest light refracted through a wavy surface.

Lightmapping

In real life it's rare to find a room that is illuminated entirely by a light source at the viewer's position, but neither are all scenes lit entirely by ambient light. Just as often, lighting comes from multiple light sources scattered about in various positions, some of them bright and some of them not so bright, some of them stationary and some of them moving. Your living room, for instance, may be illuminated by two or more table lamps, and possibly by an overhead lamp as well. And even when these sources have been turned off, light may pour in from windows or nearby rooms. All of these sources of illumination may periodically change from dim to bright and back again and may even move around in the course of the day. (The light streaming through the window, for instance, may enter at one angle in the morning and another in the evening.) If our ray-casting engine is to accurately model the kind of illumination found in the real world, it will have to model the distribution of variable local light sources throughout the maze. And the best way I know to model this distribution is through the use of lightmaps.

Just as a heightmap specifies the floor height at various points throughout the maze, a lightmap specifies lighting intensities. This can be done for every pixel in the maze or it can be done on the level of the maze grid. The first system, which

is the simpler of the two, could be called block-aligned lightmapping, by analogy to the block-aligned heightmaps we talked about in chapter 10. And the lightmaps themselves look a lot like heightmaps. Here are the lightmaps we'll use in our demo program:

```
map_type floorlites={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0,16,16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0,16,16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {32,32,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0,32,32,32, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0,32,32,32, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0,32,32,32, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0,32, 0, 0, 0, 0, 0, 0}
};

map_type ceilinglites={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0,32,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0,32,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,32,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0,32,32, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0,32, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0,32, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0,32, 0, 0, 0, 0, 0, 0}
};
```

As you can probably tell from the names *floorlites* and *ceilinglites,* the first of these is a map of light intensities for the floor and the second is a map of light intensities for the ceiling. The *floorlites[]* array will also contain light intensities for the walls, since we'll be using block-aligned walls that cannot coexist in the same maze squares with visible floor tiles, so by combining the wall and floor maps we'll avoid some potential redundancy. Pixels in wall, floor, and ceiling tiles

will have the *floorlites[]* value for the current maze square added to the intensity value of all other light sources falling on them (with a top intensity value of 32, as before).

In my continuing attempt to set a new world record for parameter passing, the prototype for our lightmapped version of the *draw_maze()* function looks like this:

```
void draw_maze(map_type map,map_type floor,map_type ceiling,
               map_type floorlites,map_type ceilinglites,
               char far *screen,int xview,int yview,
               float viewing_angle,int viewer_height,
               int ambient_level,unsigned char far *textmaps,
               unsigned char[MAXLIGHT+1][PALETTESIZE],
               unsigned char *litelevel);
```

This prototype will be placed in the file LITEMAP.H. The function itself is in the file LITEMAP.CPP.

This time, we'll use the lightsourced version of the *draw_maze()* function that we developed a few pages ago as a template for the new function. However, only three lines in the entire function need to be changed, the three lines that calculate the light level before a lightsourced pixel is added to the wall, floor, and ceiling, respectively. In the earlier function, this line looked up a color value for the current distance in the *litelevel[]* array and added the ambient term to it to find the light level of the pixel. The only additional step we need to take is to add the *floorlites[]* value for the maze square to the intensity for floor and wall pixels:

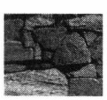
```
int level=litelevel[distance]+ambient_level+
    floorlites[xmaze][ymaze];
```

And we can add the *ceilinglites[]* value for the maze square to the intensity for ceiling pixels:

```
int level=litelevel[distance]+ambient_level+
         ceilinglites[xmaze][ymaze];
if (level>MAXLIGHT) level=MAXLIGHT;
screen[offset]=litesource[level][textmaps[tileptr]];
```

The Lightmapped draw_maze() Function

Although it seems almost redundant to include it, the text of the lightmapped version of the *draw_maze()* function appears in Listing 11-4.



Listing 11-4 The lightmapped draw_maze function

```
// LITEMAP.CPP
//
```

```
// Function to draw texture-mapped walls, floors and
// ceilings using ray casting.
// Written by Christopher Lampton for
// GARDENS OF IMAGINATION (Waite Group Press)
//

#include <stdio.h>
#include <math.h>
#include "litemap.h"
#include "pcx.h"

// Constant definitions:
const WALL_HEIGHT=64;           // Height of wall in pixels
const VIEWER_DISTANCE=192;      // Viewer distance from screen
const VIEWPORT_LEFT=0;          // Dimensions of viewport
const VIEWPORT_RIGHT=319;
const VIEWPORT_TOP=0;
const VIEWPORT_BOT=199;
const VIEWPORT_HEIGHT=VIEWPORT_BOT-VIEWPORT_TOP;
const VIEWPORT_CENTER=VIEWPORT_TOP+VIEWPORT_HEIGHT/2;
const GRIDSIZE=16;
const MAXDISTANCE=64*GRIDSIZE;

void draw_maze(map_type map,map_type floor,map_type ceiling,
                map_type floorlites,map_type ceilinglites,
                char far *screen,int xview,int yview,
                float viewing_angle,int viewer_height,
                int ambient_level,unsigned char far * textmaps,
                unsigned char litesource[MAXLIGHT+1][PALETTESIZE],
                unsigned char *litelevel)

// Draws a ray-cast image in the viewport of the maze represented
// in array MAP[], as seen from position XVIEW, YVIEW by a
// viewer looking at angle VIEWING_ANGLE where angle 0 is due
// north. (Angles are measured in radians.)

{
  // Variable declarations:
  int sy,offset;       // Pixel y position and offset
  float xd,yd;         // Distance to next wall in x and y
  int grid_x,grid_y;   // Coordinates of x and y grid lines
  float xcross_x,xcross_y; // Ray intersection coordinates
  float ycross_x,ycross_y;
  unsigned int xdist,ydist; // Distance to x and y grid lines
  int xmaze,ymaze;     // Map location of ray collision
  int distance;  // Distance to wall along ray
  int tmcolumn;        // Column in texture map
  float yratio;

  // Loop through all columns of pixels in viewport:
  for (int column=VIEWPORT_LEFT; column<VIEWPORT_RIGHT; column++) {

    // Calculate horizontal angle of ray relative to
```

continued on next page

continued from previous page

```
    //  center ray:
    float column_angle=atan((float)(column-160)
                          / VIEWER_DISTANCE);

    // Calculate angle of ray relative to maze coordinates
    float radians=viewing_angle+column_angle;

    // Rotate endpoint of ray to viewing angle:
    int x2 = -1024 * (sin(radians));
    int y2 = 1024 * (cos(radians));

    // Translate relative to viewer's position:
    x2+=xview;
    y2+=yview;

    // Initialize ray at viewer's position:
    float x=xview;
    float y=yview;

    // Find difference in x,y coordinates along ray:
    int xdiff=x2-xview;
    int ydiff=y2-yview;

    // Cheat to avoid divide-by-zero error:
    if (xdiff==0) xdiff=1;

    // Get slope of ray:
    float slope = (float)ydiff/xdiff;

    // Cheat (again) to avoid divide-by-zero error:
    if (slope==0.0) slope=.0001;

    // Cast ray from grid line to grid line:
    for (;;) {

      // If ray direction positive in x, get next x grid line:
      if (xdiff>0) grid_x=((int)x & 0xffc0)+64;

      // If ray direction negative in x, get last x grid line:
      else grid_x=((int)x & 0xffc0) - 1;

      // If ray direction positive in y, get next y grid line:
      if (ydiff>0) grid_y=((int)y & 0xffc0) +64;

      // If ray direction negative in y, get last y grid line:
      else grid_y=((int)y & 0xffc0) - 1;

      // Get x,y coordinates where ray crosses x grid line:
      xcross_x=grid_x;
      xcross_y=y+slope*(grid_x-x);

      // Get x,y coordinates where ray crosses y grid line:
```

```
    ycross_x=x+(grid_y-y)/slope;
    ycross__y=grid_y;

    // Get distance to x grid line:
    xd=xcross_x-x;
    yd=xcross_y-y;
    xdist=sqrt(xd*xd+yd*yd);

    // Get distance to y grid line
    xd=ycross_x-x;
    yd=ycross_y-y;
    ydist=sqrt(xd*xd+yd*yd);

    // If x grid line is closer...
    if (xdist<ydist) {

      // Calculate maze grid coordinates of square:
      xmaze=xcross_x/64;
      ymaze=xcross_y/64;

      // Set x and y to point of ray intersection:
      x=xcross_x;
      y=xcross_y;

      // Find relevant column of texture map
      tmcolumn = (int)y & 0x3f;

      // Is there a maze cube here? If so, stop looping:
      if (map[xmaze][ymaze]) break;
    }
    else { // If y grid line is closer:

      // Calculate maze grid coordinates of square:
      xmaze=ycross_x/64;
      ymaze=ycross_y/64;

      // Set x and y to point of ray intersection:
      x=ycross_x;
      y=ycross_y;

      // Find relevant column of texture map:
      tmcolumn = (int)x & 0x3f;

      // Is there a maze cube here? If so, stop looping:
      if (map[xmaze][ymaze]) break;
    }
  }

  // Get distance from viewer to intersection point:
  xd=x-xview;
  yd=y-yview;
```

continued on next page

continued from previous page

```
    distance=(long)sqrt(xd*xd+yd*yd)*cos(column_angle);
    if (distance==0) distance=1;

    // Calculate visible height of wall:
    int height = VIEWER_DISTANCE * WALL_HEIGHT / distance;

    // Calculate bottom of wall on screen:
    int bot = VIEWER_DISTANCE * viewer_height
              / distance + VIEWPORT_CENTER;

    // Calculate top of wall on screen:
    int top = bot - height;

    // Initialize temporary offset into texture map:
    int t=tmcolumn;

    // If top of current vertical line is outside of
    // viewport, clip it:
    int dheight=height;
    int iheight=IMAGE_HEIGHT;
    yratio=(float)WALL_HEIGHT/height;
    if (top < VIEWPORT_TOP) {
      dheight-=(VIEWPORT_TOP - top);
      t+=(int)((VIEWPORT_TOP-top)*yratio)*320;
      iheight -= ((VIEWPORT_TOP-top)*yratio);
      top=VIEWPORT_TOP;
    }
    if (bot > VIEWPORT_BOT) {
      dheight -= (bot - VIEWPORT_BOT);
      iheight -= (bot - VIEWPORT_BOT)*yratio;
      bot=VIEWPORT_BOT;
    }

    // Point to video memory offset for top of line:
    offset = top * 320 + column;

    // Initialize vertical error term for texture map:
    int tyerror=64;

    // Which graphics tile are we using?
    int tile=map[xmaze][ymaze]-1;

    // Find offset of tile and column in bitmap:
    unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
      *IMAGE_WIDTH+t;

    // Loop through the pixels in the current vertical
    // line, advancing OFFSET to the next row of pixels
    // after each pixel is drawn.
    for (int h=0; h<iheight; h++) {

      // Are we ready to draw a pixel?
```

```
  while (tyerror>=IMAGE_HEIGHT) {

    // If so, draw it:
    int level=litelevel[distance]+ambient_level+
        floorlites[xmaze][ymaze];
    if (level>MAXLIGHT) level=MAXLIGHT;
    screen[offset]=litesource[level][textmaps[tileptr]];

    // Reset error term:
    tyerror-=IMAGE_HEIGHT;

    // And advance OFFSET to next screen line:
    offset+=320;
  }

  // Incremental division:
  tyerror+=height;

  // Advance TILEPTR to next line of bitmap:
  tileptr+=320;
}

// Step through floor pixels:
for (int row=bot+1; row<=VIEWPORT_BOT; row++) {

  // Get ratio of viewer's height to pixel height:
  float ratio=(float)viewer_height/(row-100);

  // Get distance to visible pixel:
  distance=ratio*VIEWER_DISTANCE/cos(column_angle);

  // Rotate distance to ray angle:
  int x = - distance * (sin(radians));
  int y = distance * (cos(radians));

  // Translate relative to viewer coordinates:
  x+=xview;
  y+=yview;

  // Get maze square intersected by ray:
  int xmaze = x / 64;
  int ymaze = y / 64;

  // Find relevant column of texture map:
  int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);

  // Which graphics tile are we using?
  int tile=floor[xmaze][ymaze];

  // Find offset of tile and column in bitmap:
  unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
    *IMAGE_WIDTH+t;
```

continued on next page

continued from previous page

```
      // Calculate video offset of floor pixel:
      offset=row*320+column;

      // Draw pixel:
      int level=litelevel[distance]+ambient_level+
              floorlites[xmaze][ymaze];
      if (level>MAXLIGHT) level=MAXLIGHT;
      screen[offset]=litesource[level][textmaps[tileptr]];
    }

    // Step through ceiling pixels:
    for (row=top-1; row>=VIEWPORT_TOP; --row) {

      // Get ratio of viewer's height to pixel height:
      float ratio=(float)(WALL_HEIGHT-viewer_height)/(100-row);

      // Get distance to visible pixel:
      distance=ratio*VIEWER_DISTANCE/cos(column_angle);

      // Rotate distance to ray angle:
      int x = - distance * (sin(radians));
      int y = distance * (cos(radians));

      // Translate relative to viewer coordinates:
      x+=xview;
      y+=yview;

      // Get maze square intersected by ray:
      int xmaze = x / 64;
      int ymaze = y / 64;

      // Find relevant column of texture map:
      int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);

      // Which graphics tile are we using?
      int tile=ceiling[xmaze][ymaze];

      // Find offset of tile and column in bitmap:
      unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
        *IMAGE_WIDTH+t;

      // Calculate video offset of floor pixel:
      offset=row*320+column;

      // Draw pixel:
      int level=litelevel[distance]+ambient_level+
               ceilinglites[xmaze][ymaze];
      if (level>MAXLIGHT) level=MAXLIGHT;
      screen[offset]=litesource[level][textmaps[tileptr]];
    }
  }
}
```

The MAPDEMO.CPP Program

The text of MAPDEMO.CPP, a short demonstration program that calls the lightmapped version of the *draw_maze()* function to draw a lightmapped view of a maze, appears in Listing 11-5.

Listing 11-5 The MAPDEMO.CPP program

```
// MAPDEMO.CPP
//
// Calls ray-casting function to draw light-mapped
// view of maze.
//
// Written by Christopher Lampton for
// Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "screen.h"
#include  "pcx.h"
#include  "litemap.h"

const GRIDSIZE=16;
const MAXDISTANCE=64*GRIDSIZE;
const NUM_IMAGES=15;
const float MULTIPLIER=3;

pcx_struct textmaps;
FILE *handle;
unsigned char litetable[MAXLIGHT+1][PALETTESIZE];
unsigned char *litelevel;

// Map of wall textures:
map_type walls={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 0, 5, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 5},
 { 5, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 5},
 { 5, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 5},
 { 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 4, 0, 0, 0, 5},
 {12, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,11, 0, 0, 0, 5},
 {11, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,10, 0, 0, 5},
 {11, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,10, 0, 0, 5},
 {11, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0,11,10, 0, 0, 5},
 {11, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,11, 0, 0, 0, 5},
 {12, 5, 5, 0, 0, 0, 5, 1, 0, 1, 0, 0, 0, 0, 0, 5},
 { 5, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 5},
```

continued on next page

continued from previous page

```
 { 5, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,12},
 { 5, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,11},
 { 5, 6, 6, 6, 6, 6, 0, 0, 1, 0, 1, 0, 0, 0, 0,10},
 { 5, 5, 5, 5, 5, 5, 5,15,11,15, 5, 5, 5, 5, 5, 5}
};

// Map of floor textures:
map_type flor={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 8, 8, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 8, 8, 8, 8, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 8, 8, 8, 5, 5, 5, 5, 5, 8, 5, 5, 5, 5},
 { 5, 5, 5, 5, 8, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
};

// Map of ceiling textures:
map_type ceiling={
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,12,13,13,13,13},
 {13,13,13, 6, 6,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13, 6, 6,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13, 6, 6,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13}
};

// Map of floor lighting:
map_type floorlites={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

```
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0,16,16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0,16,16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {32,32,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0,32,32,32, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0,32,32,32, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0,32,32,32, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0,32, 0, 0, 0, 0, 0, 0, 0}
};

// Map of ceiling lighting:
map_type ceilinglites={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0,32,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0,32,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0,32,32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0,32,32, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0,32, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0,32, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0,32, 0, 0, 0, 0, 0, 0, 0}
};

float viewing_angle=0;
float intensity=MAXLIGHT;
int viewer_height=32;
int ambient_level=0;
int xview=8*64+32;
int yview=8*64+32;
int lighttype=1;

void main(int argc,char* argv[])
{
  // Read arguments from command line if present:
  if (argc>=2) viewing_angle=atof(argv[1]);
  if (argc>=3) intensity=atof(argv[2]);
  if (argc>=4) ambient_level=atof(argv[3]);

  // Load texture map images:
  if (loadPCX("images3.pcx",&textmaps)) exit(1);

  // Load lightsourcing tables:
  if ((handle=fopen("litesorc.dat","rb"))==NULL) {
```

continued on next page

continued from previous page

```
    perror("Error");
    exit;
  }
  fread(litetable,MAXLIGHT+1,PALETTESIZE,handle);
  fclose(handle);

  // Allocate memory for array of light levels:
  litelevel=new unsigned char[MAXDISTANCE];

  // Calculate light levels by distance:
  for (int distance=1; distance<MAXDISTANCE; distance++) {
    float ratio=(float)intensity/distance*MULTIPLIER;
    if (ratio>1.0) ratio=1.0;
    litelevel[distance]=ratio*MAXLIGHT;
  }

  // Point variable at video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);

  // Set mode 13h:
  setmode(0x13);

  // Set the palette:
  setpalette(textmaps.palette);

  // Clear the screen:
  cls(screen);

  // Draw a ray-cast view of the maze:
  draw_maze(walls,flor,ceiling,floorlites,ceilinglites,
              screen,xview,yview,viewing_angle,viewer_height,
              ambient_level,textmaps.image,litetable,
              litelevel);

  // Wait for user to hit a key:
  while (!kbhit());

  // Release memory
  delete textmaps.image;
  delete litelevel;

  // Reset video mode and exit:
  setmode(oldmode);
```

To run the program, go to the LIGHT directory and type MAPDEMO. The result is shown in Figure 11-11. Several portions of floors, walls, and ceilings have been illuminated, an effect that becomes more obvious in the distance, where the light from the player's torch becomes less intense. As in the previous demo, you can type in three command line arguments, representing the angle of

view, the intensity of the light carried by the viewer, and the ambient lighting level. If you type MAPDEMO 3.14 0 0 you'll see the same view as above, but with the player's light turned completely off. Now the lightmapping is the *only* source of illumination, an effect that is quite dramatic but not entirely realistic. In the real world, at least *some* light would spill over from the illuminated areas into the rest of the scene. We can simulate this effect by raising the ambient level ever so slightly, with the instruction MAPDEMO 3.14 0 4 (see Figure 11-12). Or we can raise the intensity of the torch instead, with the instruction MAPDEMO 3.14 4 0, as shown in Figure 11-13. Play around with different angles, source intensities, and ambient levels for a while and see what sort of effects you can produce.

Tiled Lightmaps

The limitations of block-aligned lightmaps are pretty much the same as the limitations of block-aligned heightmaps. Changes in light intensity can only occur at block boundaries and must therefore have squared off edges, an effect that isn't always realistic. Just as tiled heightmaps helped us solve some of the problems of block-aligned heightmapping, tiled lightmaps can help us solve the

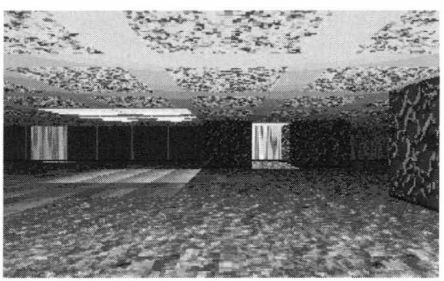


Figure 11-11 Lightmap tiles in a maze otherwise lit by the viewer's torch

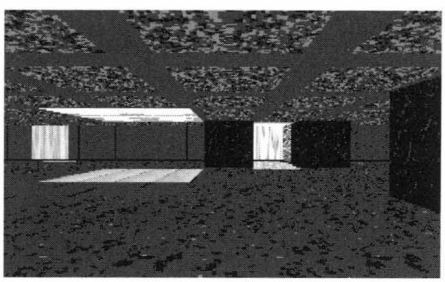


Figure 11-12 Lightmap tiles glowing in the middle of a pitch-dark maze

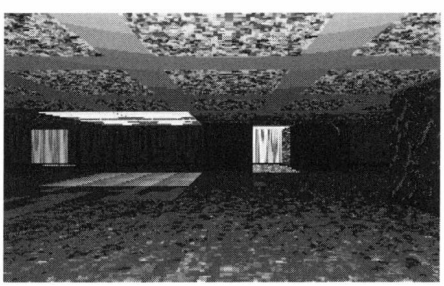


Figure 11-13 Lightmap tiles in a slightly better illuminated maze

problems of block-aligned lightmapping. And tiled lightmaps are easier to use than tiled heightmaps, since we don't have to worry about raised pixels blocking our view of height changes that fall partially in their shadow.

By now, you can probably figure out how tiled lightmaps work without any prompting from me. There's a PCX file containing multiple lightmap tiles in the LIGHT directory under the name LITEMAPS.PCX. It consists of a nine-part map forming a large circular pool of light (the middle tile of which doubles as a fully illuminated square of light), a single tile representing a small circular pool of light, and a tile with a diagonal slash of light. Each pixel in the lightmaps has a color value in the range 0 to 33, representing the amount to be added to the intensity values of the corresponding screen pixels, the total value not to exceed 32.

We'll still pass the *floorlites[]* and *ceilinglites[]* arrays to the *draw_maze()* function. And — for yet another world record — we'll pass an additional parameter called *litemaps* that points to the light tiles themselves. The new prototype, in the file TLMAP.H, looks like this:

```
void draw_maze(map_type map,map_type floor,map_type ceiling,
                map_type floorlites,map_type ceilinglites,
                char far *screen,int xview,int yview,
                float viewing_angle,int viewer_height,
                int ambient_level,unsigned char far *textmaps,
                unsigned char far *litemaps,
                unsigned char[MAXLIGHT+1][PALETTESIZE],
                unsigned char *litelevel);
```

Before we place a pixel on the display, we've been calculating a value called *tileptr* to point to the offset in the texture-map array that contains the color value for that pixel. Now we'll use exactly the same method that we used to calculate *tileptr* to calculate a value that we'll call *liteptr*, which will point to the value in the lightmap array that represents the light intensity of the current pixel:

```
tile=floorlites[xmaze][ymaze];
unsigned int liteptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;
```

Then we can add this value to the intensity value for the pixel:

```
int level=litelevel[distance]+ambient_level+
      litemaps[liteptr];
```

And that's all the code that we need to add to use tiled lightmaps.

The Tiled Lightmapping draw_maze() Function

The complete text of the tiled lightmapping version of *draw_maze()* appears in Listing 11-6.

Listing 11-6 The tiled lightmapping draw_maze function

```
// LITEMAP.CPP
//
// Function to draw texture-mapped walls, floors and
// ceilings using ray casting.
// Written by Christopher Lampton for
// GARDENS OF IMAGINATION (Waite Group Press).
//

#include <stdio.h>
#include <math.h>
#include "tlmap.h"
#include "pcx.h"

// Constant definitions:
const WALL_HEIGHT=64;           // Height of wall in pixels
const VIEWER_DISTANCE=192;      // Viewer distance from screen
const VIEWPORT_LEFT=0;          // Dimensions of viewport
const VIEWPORT_RIGHT=319;
const VIEWPORT_TOP=0;
const VIEWPORT_BOT=199;
const VIEWPORT_HEIGHT=VIEWPORT_BOT-VIEWPORT_TOP;
const VIEWPORT_CENTER=VIEWPORT_TOP+VIEWPORT_HEIGHT/2;
const GRIDSIZE=16;
const MAXDISTANCE=64*GRIDSIZE;

void draw_maze(map_type map,map_type floor,map_type ceiling,
                map_type floorlites,map_type ceilinglites,
                char far *screen,int xview,int yview,
                float viewing_angle,int viewer_height,
                int ambient_level,unsigned char far *textmaps,
                unsigned char far *litemaps,
                unsigned char litesource[MAXLIGHT+1][PALETTESIZE],
                unsigned char *litelevel)

// Draws a ray-cast image in the viewport of the maze represented
// in array MAP[], as seen from position XVIEW, YVIEW by a
// viewer looking at angle VIEWING_ANGLE where angle 0 is due
// north. (Angles are measured in radians.)

{
  // Variable declarations:
  int sy,offset;       // Pixel y position and offset
  float xd,yd;         // Distance to next wall in x and y
  int grid_x,grid_y;   // Coordinates of x and y grid lines
  float xcross_x,xcross_y; // Ray intersection coordinates
```

continued on next page

continued from previous page

```
  float ycross_x,ycross_y;
  unsigned int xdist,ydist; // Distance to x and y grid lines
  int xmaze,ymaze;     // Map location of ray collision
  int distance;  // Distance to wall along ray
  int tmcolumn;        // Column in texture map
  float yratio;

  // Loop through all columns of pixels in viewport:
  for (int column=VIEWPORT_LEFT; column<VIEWPORT_RIGHT; column++) {

    // Calculate horizontal angle of ray relative to
    //  center ray:
    float column_angle=atan((float)(column-160)
                        / VIEWER_DISTANCE);

    // Calculate angle of ray relative to maze coordinates
    float radians=viewing_angle+column_angle;

    // Rotate endpoint of ray to viewing angle:
    int x2 = -1024 * (sin(radians));
    int y2 = 1024 * (cos(radians));

    // Translate relative to viewer's position:
    x2+=xview;
    y2+=yview;

    // Initialize ray at viewer's position:
    float x=xview;
    float y=yview;

    // Find difference in x,y coordinates along ray:
    int xdiff=x2-xview;
    int ydiff=y2-yview;

    // Cheat to avoid divide-by-zero error:
    if (xdiff==0) xdiff=1;

    // Get slope of ray:
    float slope = (float)ydiff/xdiff;

    // Cheat (again) to avoid divide-by-zero error:
    if (slope==0.0) slope=.0001;

    // Cast ray from grid line to grid line:
    for (;;) {

      // If ray direction positive in x, get next x grid line:
      if (xdiff>0) grid_x=((int)x & 0xffc0)+64;

      // If ray direction negative in x, get last x grid line:
      else grid_x=((int)x & 0xffc0) - 1;

      // If ray direction positive in y, get next y grid line:
      if (ydiff>0) grid_y=((int)y & 0xffc0) +64;
```

```
// If ray direction negative in y, get last y grid line:
else grid_y=((int)y & 0xffc0) - 1;

// Get x,y coordinates where ray crosses x grid line:
xcross_x=grid_x;
xcross_y=y+slope*(grid_x-x);

// Get x,y coordinates where ray crosses y grid line:
ycross_x=x+(grid_y-y)/slope;
ycross_y=grid_y;

// Get distance to x grid line:
xd=xcross_x-x;
yd=xcross_y-y;
xdist=sqrt(xd*xd+yd*yd);

// Get distance to y grid line:
xd=ycross_x-x;
yd=ycross_y-y;
ydist=sqrt(xd*xd+yd*yd);

// If x grid line is closer...
if (xdist<ydist) {

  // Calculate maze grid coordinates of square:
  xmaze=xcross_x/64;
  ymaze=xcross_y/64;

  // Set x and y to point of ray intersection:
  x=xcross_x;
  y=xcross_y;

  // Find relevant column of texture map:
  tmcolumn = (int)y & 0x3f;

  // Is there a maze cube here? If so, stop looping:
  if (map[xmaze][ymaze]) break;
}
else { // If y grid line is closer:

  // Calculate maze grid coordinates of square:
  xmaze=ycross_x/64;
  ymaze=ycross_y/64;

  // Set x and y to point of ray intersection:
  x=ycross_x;
  y=ycross_y;

  // Find relevant column of texture map:
  tmcolumn = (int)x & 0x3f;

  // Is there a maze cube here? If so, stop looping:
  if (map[xmaze][ymaze]) break;
}
```

continued on next page

continued from previous page

```
}

// Get distance from viewer to intersection point:
xd=x-xview;
yd=y-yview;
distance=(long)sqrt(xd*xd+yd*yd)*cos(column_angle);
if (distance==0) distance=1;

// Calculate visible height of wall:
int height = VIEWER_DISTANCE * WALL_HEIGHT / distance;

// Calculate bottom of wall on screen:
int bot = VIEWER_DISTANCE * viewer_height
           / distance + VIEWPORT_CENTER;

// Calculate top of wall on screen:
int top = bot - height;

// Initialize temporary offset into texture map:
int t=tmcolumn;

// If top of current vertical line is outside of
// viewport, clip it:
int dheight=height;
int iheight=IMAGE_HEIGHT;
yratio=(float)WALL_HEIGHT/height;
if (top < VIEWPORT_TOP) {
  dheight-=(VIEWPORT_TOP - top);
  t+=(int)((VIEWPORT_TOP-top)*yratio)*320;
  iheight -= ((VIEWPORT_TOP-top)*yratio);
  top=VIEWPORT_TOP;
}
if (bot > VIEWPORT_BOT) {
  dheight -= (bot - VIEWPORT_BOT);
  iheight -= (bot - VIEWPORT_BOT)*yratio;
  bot=VIEWPORT_BOT;
}

// Point to video memory offset for top of line:
offset = top * 320 + column;

// Initialize vertical error term for texture map:
int tyerror=64;

// Which graphics tile are we using?
int tile=map[xmaze][ymaze]-1;

// Find offset of tile and column in bitmap:
unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;

// Which light tile are we using?
tile=floorlites[xmaze][ymaze];
```

```
// Find offset of tile and column in bitmap:
unsigned int liteptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;

// Loop through the pixels in the current vertical
// line, advancing OFFSET to the next row of pixels
// after each pixel is drawn.
for (int h=0; h<iheight; h++) {

  // Are we ready to draw a pixel?
  while (tyerror>=IMAGE_HEIGHT) {

    // If so, draw it:
    int level=litelevel[distance]+ambient_level+
        litemaps[liteptr];
    if (level>MAXLIGHT) level=MAXLIGHT;
      screen[offset]=litesource[level][textmaps[tileptr]];

    // Reset error term:
    tyerror-=IMAGE_HEIGHT;

    // And advance OFFSET to next screen line:
    offset+=320;
  }

  // Incremental division:
  tyerror+=height;

  // Advance TILEPTR * LITEPTR to next line of bitmap:
  tileptr+=320;
  liteptr+=320;
}

// Step through floor pixels:
for (int row=bot+1; row<=VIEWPORT_BOT; row++) {

  // Get ratio of viewer's height to pixel height:
  float ratio=(float)viewer_height/(row-100);

  // Get distance to visible pixel:
  distance=ratio*VIEWER_DISTANCE/cos(column_angle);

  // Rotate distance to ray angle:
  int x = - distance * (sin(radians));
  int y = distance * (cos(radians));

  // Translate relative to viewer coordinates:
  x+=xview;
  y+=yview;

  // Get maze square intersected by ray:
  int xmaze = x / 64;
  int ymaze = y / 64;
```

continued on next page

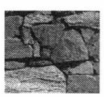
continued from previous page

```
      // Find relevant column of texture map:
      int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);

      // Which graphics tile are we using?
      int tile=floor[xmaze][ymaze];

      // Find offset of tile and column in bitmap:
      unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
        *IMAGE_WIDTH+t;

      // Which light tile are we using?
      tile=floorlites[xmaze][ymaze];

      // Find offset of tile and column in bitmap:
      unsigned int liteptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
        *IMAGE_WIDTH+t;

      // Calculate video offset of floor pixel:
      offset=row*320+column;

      // Draw pixel:
      int level=litelevel[distance]+ambient_level+
              litemaps[liteptr];
      if (level>MAXLIGHT) level=MAXLIGHT;
      screen[offset]=litesource[level][textmaps[tileptr]];
    }

    // Step through ceiling pixels:
    for (row=top-1; row>=VIEWPORT_TOP; --row) {

      // Get ratio of viewer's height to pixel height:
      float ratio=(float)(WALL_HEIGHT-viewer_height)/(100-row);

      // Get distance to visible pixel:
      distance=ratio*VIEWER_DISTANCE/cos(column_angle);

      // Rotate distance to ray angle:
      int x = - distance * (sin(radians));
      int y = distance * (cos(radians));

      // Translate relative to viewer coordinates:
      x+=xview;
      y+=yview;

      // Get maze square intersected by ray:
      int xmaze = x / 64;
      int ymaze = y / 64;

      // Find relevant column of texture map:
      int t = ((int)y & 0x3f) * 320 + ((int)x & 0x3f);

      // Which graphics tile are we using?
```

```
      int tile=ceiling[xmaze][ymaze];

      // Find offset of tile and column in bitmap:
      unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
        *IMAGE_WIDTH+t;

      // Which light tile are we using?
      tile=ceilinglites[xmaze][ymaze];

      // Find offset of tile and column in bitmap:
      unsigned int liteptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
        *IMAGE_WIDTH+t;

      // Calculate video offset of floor pixel:
      offset=row*320+column;

      // Draw pixel:
      int level=litelevel[distance]+ambient_level+
            litemaps[liteptr];
      if (level>MAXLIGHT) level=MAXLIGHT;
      screen[offset]=litesource[level][textmaps[tileptr]];
    }

  }
}
```

The TLMDEMO.CPP Program

To demonstrate the tiled lightmapping version of the *draw_maze()* function, we'll use the simple program in Listing 11-7.



Listing 11-7 The TLMDEMO.CPP program

```
// TLMDEMO.CPP
//
// Calls ray-casting function to draw tiled light-mapped
// view of maze.
//
// Written by Christopher Lampton for
// Gardens of Imagination (Waite Group Press)

#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "screen.h"
#include  "pcx.h"
```

continued on next page

continued from previous page

```
#include  "tlmap.h"

const GRIDSIZE=16;
const MAXDISTANCE=64*GRIDSIZE;
const NUM_IMAGES=15;
const float MULTIPLIER=3;

pcx_struct textmaps,litemaps;
FILE *handle;
unsigned char litetable[MAXLIGHT+1][PALETTESIZE];
unsigned char *litelevel;

// Map of wall textures:
map_type walls={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 0, 5, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 5},
 { 5, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 5},
 { 5, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 5},
 { 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 4, 0, 0, 0, 5},
 {12, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,11, 0, 0, 0, 5},
 {11, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,10, 0, 0, 5},
 {11, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,10, 0, 0, 5},
 {11, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0,11,10, 0, 0, 5},
 {11, 0, 0, 0, 0, 0, 0, 0, 0, 0,11,11, 0, 0, 0, 5},
 {12, 5, 5, 0, 0, 0, 5, 1, 0, 1, 0, 0, 0, 0, 0, 5},
 { 5, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 5},
 { 5, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,12},
 { 5, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,11},
 { 5, 6, 6, 6, 6, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,10},
 { 5, 5, 5, 5, 5, 5, 5,11,11,11, 5, 5, 5, 5, 5, 5}
};

// Map of floor textures:
map_type flor={
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 8, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5},
 { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
};

// Map of ceiling textures:
```

```
map_type ceiling={
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,12,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13},
 {13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13}
};

// Map of floor lighting:
map_type floorlites={
 { 0, 0, 0, 0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 2, 7,12, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 3, 8,13, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 1, 6,11, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 2, 7,12, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 3, 8,13, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 7, 7, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 7, 4, 0, 0, 0, 0, 0, 0, 0}
};

// Map of ceiling lighting:
map_type ceilinglites={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

continued on next page

continued from previous page

```
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};

float viewing_angle=0;
float intensity=MAXLIGHT;
int viewer_height=32;
int ambient_level=0;
int xview=8*64+32;
int yview=8*64+32;
int lighttype=1;

void main(int argc,char* argv[])
{

  // Read arguments from command line if present:
  if (argc>=2) viewing_angle=atof(argv[1]);
  if (argc>=3) intensity=atof(argv[2]);
  if (argc>=4) ambient_level=atof(argv[3]);

  // Load texture map images:
  if (loadPCX("images3.pcx",&textmaps)) exit(1);

  // Load light map tiles:
  if (loadPCX("litemaps.pcx",&litemaps)) exit(1);

  // Load lightsourcing tables:
  if ((handle=fopen("litesorc.dat","rb"))==NULL) {
    perror("Error");
    exit;
  }
  fread(litetable,MAXLIGHT+1,PALETTESIZE,handle);
  fclose(handle);

  // Allocate memory for array of light levels:
  litelevel=new unsigned char[MAXDISTANCE];

  // Calculate light levels by distance:
  for (int distance=1; distance<MAXDISTANCE; distance++) {
    float ratio=(float)intensity/distance*MULTIPLIER;
    if (ratio>1.0) ratio=1.0;
    litelevel[distance]=ratio*MAXLIGHT;
  }

  // Point variable at video memory:
  char far *screen=(char far *)MK_FP(0xa000,0);

  // Save previous video mode:
  int oldmode=*(int *)MK_FP(0x40,0x49);
```

```
  // Set mode 13h:
  setmode(0x13);

  // Set the palette:
  setpalette(textmaps.palette);

  // Clear the screen:
  cls(screen);

  // Draw a ray-cast view of the maze:
  draw_maze(walls,flor,ceiling,floorlites,ceilinglites,
              screen,xview,yview,viewing_angle,viewer_height,
              ambient_level,textmaps.image,litemaps.image,
              litetable,litelevel);

  // Wait for user to hit a key:
  while (!kbhit());

  // Release memory
  delete textmaps.image;
  delete litemaps.image;
  delete litelevel;

  // Reset video mode and exit:
  setmode(oldmode);
}
```

You can use this program by going to the LIGHT directory and typing TLMDEMO. The result, with a large circular pool of light visible in the distance, is shown in Figure 11-14. As before, you can pass the program command line parameters representing the viewing angle, the intensity of the viewer's torch, and the ambient lighting level. Look around, experiment a little, and see what works and what doesn't.

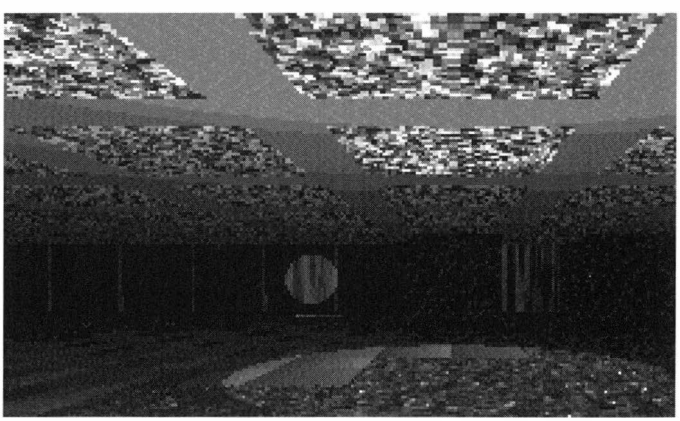


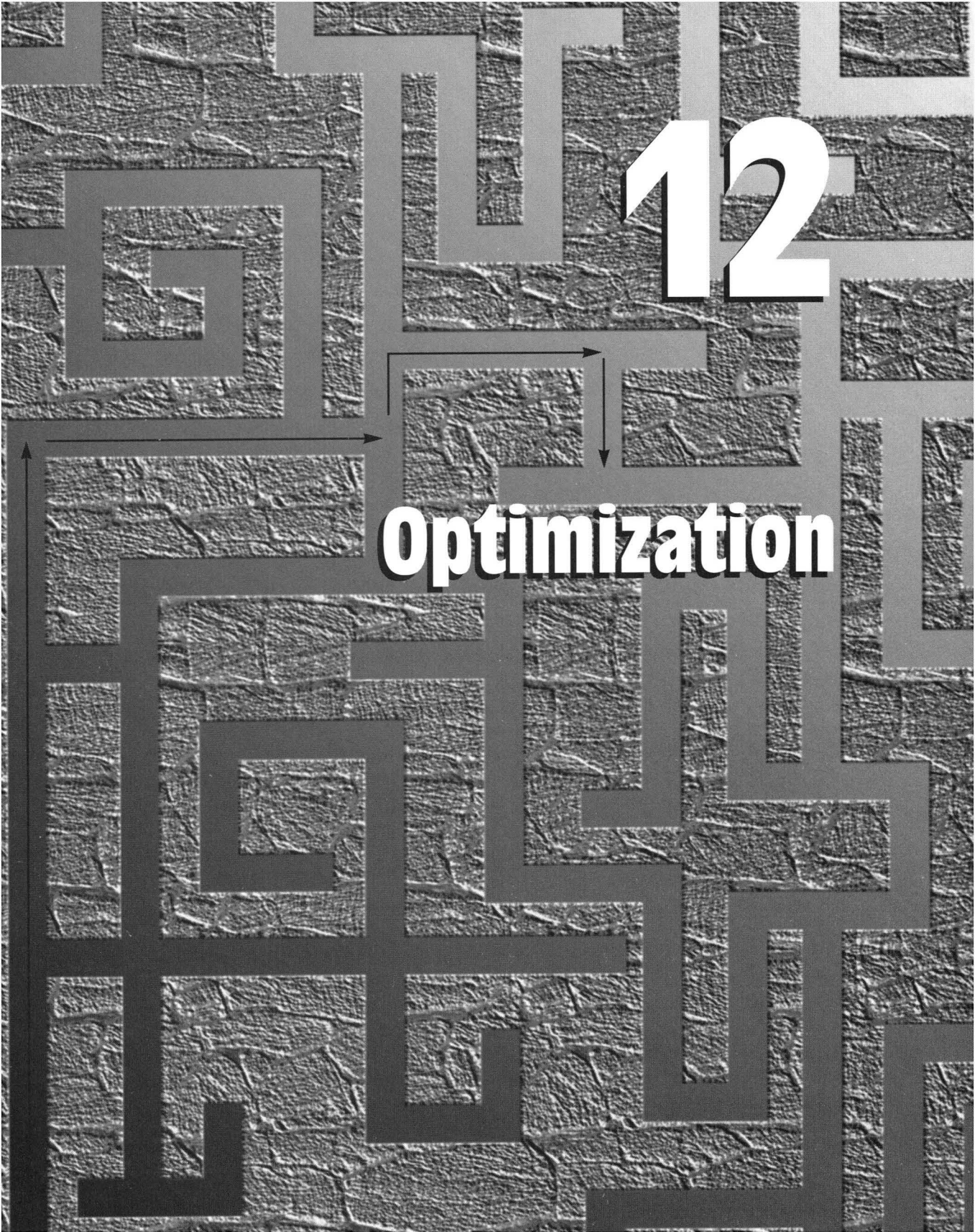
Figure 11-14 A screenshot from the TLMDEMO program

Using Lightmaps

Lightmapping can be used to create the illusion of bright overhead light sources in a dungeon, or of light spilling out of another room. Entire areas of the maze can be mapped to their maximum intensity, to give the impression that those areas are in full daylight. Animated characters, which should be drawn using the same light intensity values that have been assigned to the floor squares (or pixels) upon which they are standing, will appear lighted or shadowed depending on where they are standing. They'll even appear silhouetted if they are standing in a dimly lit area with a bright lightmap directly behind them, a very dramatic effect.

Where lightmapping comes into its own, though, is in animation. The most impressive effects of all can be achieved by changing the lightmapping from frame to frame, which is a simple matter of writing new values into the *floorlite[]* and *ceilinglite[]* arrays. Lights in the maze can be made to blink on and off, in strobe light fashion, or to oscillate between dim and bright, as though someone were turning a rheostat. Changing light values sequentially from square to square — that is, turning one square off and the next square on in sequence — can give the impression that the light source is moving. If an animated fireball or other illuminated object moves from square to square along with the changing light source, it will appear as though the object is casting the light. With block-aligned lightmaps this effect might be a little rough, with the light source seeming to jump from maze square to maze square, but a series of light tiles could be designed so that it would seem as though the lightmapping were moving only a few pixels per frame.

There are plenty of ways in which an imaginative programmer or game designer can utilize lightmapping to create dramatic visual effects. For some ideas, look at Id Software's Doom and Bethesda Softworks' The Elder Scrolls: Arena, two games that make creative use of what appears to be lightmapping. And then come up with some new ideas of your own.


12
Optimization

12

he ray-casting engines that we've developed over the last three chapters do a great job of rendering still images of the interior of a maze. But if still images were all that we wanted to create, we might as well use a ray tracer such as POVRAY, which would produce even more vivid images and render a wider variety of shapes. The whole point of using a ray-casting engine rather than a ray tracer is to produce images that *move!*

Before our ray-casting engine is capable of creating real time animation, though, we'll need to optimize it so that it will run a great deal faster. Fortunately, I have a few tricks up my sleeve that will speed up this engine to the point where it will be useful in an animated computer game. By the end of this chapter, we'll have increased the speed of the engine to the point where it will be chugging along at a more than satisfactory pace.

Mind you, the optimization tricks discussed in this chapter are not all that exist. A clever programmer should be able to speed up the engine developed in this chapter even further; don't assume that the tips and tricks mentioned in this chapter represent the whole extent of the programmer's art. Rather, the program presented in this chapter should set you on your way to creating an even more optimized program of your own.

The Target Machine

The precise way in which you optimize your program will depend on the types of computers you expect it to be running on. For instance, considerably greater optimization will be necessary if you want your program to run well on machines based on the (now ancient) 80286 processor than if you expect it to run only on Pentium machines, since even the slowest Pentium processors are many times faster than the fastest 286s. And, as we'll see below, there are special quirks of the 486 and Pentium processors that must be taken into account during optimization.

When you start to optimize, you should have a target machine in mind. This is the minimum computer configuration necessary to run the program at acceptable speed. You'll need to have this machine available for testing the program during the optimization process, so that you'll know when you've reached the point in your optimization where the program runs well on that machine. When that point is reached, you can stop optimizing, even if more optimization is theoretically possible.

What kind of machine should you choose for your target machine? The answer to that question depends on a lot of variables, not the least of which is the time frame in which you expect the program to be released. The optimal target machine for the spring of 1995 won't be the same as the optimal target machine for the fall of 1996 — or even the fall of 1995. Guessing what the optimal machine *will* be, however, is a tricky business.

At the time this book was written, the 80286 processor had long ago ceased to be a viable gaming machine and had been pretty much abandoned by game developers as a target machine. The same process was beginning to happen with the 80386 as well. By the time you read this, there will probably be no real need to use anything slower than an 80486 as a target machine, and you may well want to target the Pentium or whatever chip Intel releases as the Pentium's successor. Just be careful not to choose such an advanced target configuration that only a few lucky users will be able to run your game.

A Few Optimization Tricks

Most optimization tricks can be grouped into a few categories. We'll discuss those categories before we dive into the optimizations themselves.

Translating into Assembly Language

Perhaps the oldest optimization trick of all is to rewrite high-level code — code written in C++, for instance — in assembly language. Because assembly language

allows the programmer to specify the precise machine-level instructions that will make up the final program, it allows clever programmers to perform optimizations that would not be possible in high-level code. Of course, as compilers have become better at performing optimizations on their own, and as microprocessors have been redesigned in such a way that high-level optimizations are more and more effective, this optimization technique has become increasingly less important. But it still can produce effective increases in execution speed when used on time-critical code. In a moment, I'll discuss ways of identifying which portions of a program are time critical.

The Borland C++ compiler, like most state-of-the-art C/C++ compilers, lets us write certain portions of a program in assembly language and others in high-level code, so it isn't necessary to translate an entire program into assembly language. This is for the best, since translating a nontrivial program into assembly language is a major, and extremely time-consuming, task. In this chapter, we'll concentrate on translating three relatively small sections of our ray-casting engine into assembly language.

Incidentally, just because I'm translating portions of this ray-casting engine into assembly language doesn't mean that you necessarily should include assembly language modules in your own game engines. Many of the optimizations discussed in this chapter don't require assembly language in order to produce beneficial results. If you aren't yet comfortable with assembler programming, write your first games in straight C++ or whatever high-level language you feel comfortable using. Then, once you've mastered (or at least gotten pretty good at) high-level game programming, consider getting a good book on 80x86 assembly language, such as Robert Lafore's *Assembly Language Primer* from The Waite Group, and translating selected portions of your C++ code into assembler as a learning exercise. But don't do this until you feel ready.

Unrolling the Loop

Loops are one of the most important techniques in the programmer's bag of tricks. By instructing the computer to execute a block of code a fixed number of times, or until some event occurs, a lot of power can be packed into a relatively small number of programming lines. And by changing the value of the variables referenced within the loop, the same block of code can perform a different task on each iteration.

But there's a drawback to loops, especially when the code of which the loop is part needs to be highly optimized in order to perform acceptably. The instructions that cause the computer to repeat a block of code take time to execute. If the block of code in the loop is particularly small, these instructions may well take more time than the instructions in the loop itself. And if the loop will be executing many times a second, this extra time may be noticeable.

What can be done about this? In an extreme case — and we'll be looking at a couple of these extreme cases in a moment — we can eliminate the loop altogether. Since a loop is nothing more than a set of instructions that are executed repeatedly, we can write those instructions over and over again the number of times that they would normally be repeated. For instance, if the loop were to iterate ten times, we could replace the loop with a set of instructions repeated ten times over. This eliminates the "bookkeeping" instructions required by the loop, such as comparing values of variables against the loop's limit values. All that will be left is the code that does the *real* work, such as placing pixels on the display.

Perhaps your first response is that such an *unrolled loop* would take up a lot of space in the program. But that's not necessarily so. If we write the loop code in assembly language, the size of the code within the loop can be made quite small and the resulting unrolled loop will often be no more than a few kilobytes in length — small in comparison to the hundreds of kilobytes that the completed program and its data will take up. You might also think that an unrolled loop would take a lot of typing to create, but the use of macro instructions (not to mention the cut-and-paste capabilities of the Borland editor) can reduce the typing considerably. More about that in a moment.

There's one more thing that you should know before you unroll a few loops of your own. On the 80486 microprocessor, loop unrolling might not produce the speed gains that would normally be expected. That's because the 486 has a 4-kilobyte internal memory cache, where instructions are stored after being executed so that they can be executed more quickly the second time around. This allows short loops — that is, those under 4 kilobytes in length — to execute quite rapidly on the 486. That's not to say that further speed-ups can't be achieved by unrolling a loop or two. But if unrolling the loop makes it longer than 4 kilobytes, it will no longer be able to fit into the cache; and the extra time that the 486 spends loading the instructions back into the cache on each iteration of the loop will more than cancel out the gains.

Fortunately, it's not necessary to unroll a loop *completely* in order to optimize it. Unrolling it perhaps eight times, then placing those eight iterations in a loop, will give you almost 90 percent of the speed-up of a complete unrolling. For instance, a loop that executes 80 times can be unrolled eight times, then placed in a loop that executes ten times. We'll look at such a partially unrolled loop in a moment.

Table Look-Ups

Mathematical calculations, of which our ray-casting code has more than a few, can be quite time consuming to perform. One way to speed up the performance

of just about any piece of computation-intensive code is to avoid performing those computations while the code is running. Instead, perform them in advance — and place the results in an array of values from which they can be retrieved later, when they're needed.

This is especially useful in speeding up trigonometric calculations, such as sines and cosines. Most programs that feature three-dimensional graphic effects only need to take the sines and cosines of a relatively limited number of angles. For instance, if a program only allows rays or objects (the viewer included) to be rotated in 1-degree increments, then only 360 different sines and 360 different cosines will be needed. These can be calculated in advance and stored in 360-element arrays. Later, when the sine or cosine of a given angle is required, the angle can be used as an index into the appropriate array and the sine or cosine extracted with minimum strain on the CPU.

These tables can be calculated either during program initialization (if the task isn't *too* time consuming) or at compile time. (If calculated at compile time, the tables can either be placed in program modules and compiled along with the rest of the program, or placed in binary data files, from which they can be loaded during program initialization.)

We'll take a closer look at the precalculation of trigonometric tables in a moment. First, let's take a look at the type of arithmetic that we'll be using in these tables.

Fixed Point Arithmetic

Until now, whenever we've needed to solve some heavy mathematical problems in the program we've been developing, we've turned to floating point math to do the job. A C/C++ compiler uses floating point math when performing standard arithmetic operations, such as addition, subtraction, multiplication, or division, on pairs of numbers in which at least one number is declared as a *float,* a *double,* or a *long double.* Floating point is also used by library functions such as *sqrt(), sin(),* and *cos().* Although floating point is easy to use (because it's supported automatically by most compilers), it isn't always the fastest way to get the desired mathematical results.

When *isn't* floating point the fastest way to get results? That's a complicated question. On any machine that doesn't have a built-in floating point unit, or *fpu,* floating point arithmetic can be slower than Washington, D.C. traffic at rush hour. However, the majority of recent computers — including those with 80486DX and Pentium processors — come with floating point units as standard equipment. And even those machines that are not invariably equipped with fpu's, such as those based on the 80286, 80386, and 80486SX processors, often have an optional unit installed in a special socket on the motherboard.

Even those machines with fpu's installed can become somewhat sluggish when too much floating point math is used — with one exception: machines based on Pentium processors. The Pentium has a built-in fpu that reportedly works at lightning speed. On a Pentium, floating point may well be the best way to go when complex mathematical operations are required in a program.

At the time this book was written, Pentium processors still represented a minor portion of the PC market. Before long, though, the Pentium may be the domimant processor on the computing scene. Until that happens, it's best for programmers of high-speed animated games to avoid floating point math wherever possible.

What's the alternative to floating point math? Fixed point math, of course. Fixed point mathematics is a surprisingly simple method for performing complex mathematical operations — in particular, those that involve fractions — using the integer registers of an IBM-compatible CPU. The 32-bit registers and instructions set of the 386 and 486 microprocessors are especially amenable to fixed point operations.

I'll have more to say about fixed point arithmetic in a moment.

Where to Optimize?

Just as important as knowing *how* to optimize program code is knowing *where* to optimize the code. There's no point in translating an entire game program into assembly language — a task that may take longer than writing the high-level version of the code in the first place. (Assembly language is also more difficult to debug and maintain than high-level code, so you'll want to use it only in the most crucial places.) Not every loop needs to be unrolled; in fact, only a few need to be. There are even places in the most optimized of game programs where it's perfectly acceptable to use floating point math, even if the resulting program is targeted for a slow processor.

But any program that makes use of high-speed animation has a few time-critical junctures that would profit from highly optimized code. The question is, how do you know where those time-critical junctures are?

The answer is: Look for the innermost loops in the animation code.

If you're at all fluent in writing computer code, you're familiar with the concept of nested loops. Almost every nontrivial computer program contains a few of them: loops that are located inside other loops and that are executed several times for every time that the outer loop is executed. Most programs even contain loops that are nested inside loops that are nested inside loops.

As a rule of thumb, we can say that the more deeply nested a loop is inside other loops, the more important it is for that loop to be optimized. Why? Because inner loops are executed more times than outer loops. If a for() loop designed to execute 100 times is located inside another for() loop that also executes 100 times, which is in turn located inside yet another for() loop that executes 100 times, the innermost for() loop will execute 1 million times every time the outer loop executes 100 times. That's a lot of iterations! Optimizations in the innermost loop will pay you back many times over compared to optimizations in the outer loop. In fact, the outermost loop may not need to be optimized at all, unless it contains some unusually sluggish code.

How do you locate the innermost loops in your program? In many cases, they will be obvious. In our ray-casting engine, for instance, the innermost loops are the ones that draw the pixels in walls, the pixels in the floor, and the pixels in the ceiling. Simply by optimizing these loops, we can speed up the execution of the engine a thousandfold. Further optimizations will produce even more beneficial results, but optimizing these three inner loops will produce code that is fast enough for many animation purposes.

The *profiler* is a helpful tool in determining where a program most needs to be optimized. Profilers are programs that analyze computer code in order to determine which subprograms and even individual lines of code are being executed most often. By following the suggestions of a good profiler and optimizing only those pieces of code that the CPU spends most of its time executing, you can save a lot of time that might otherwise be spent optimizing non-time-critical portions of a program. A good profiler will help you place your programming resources where they really count.

The Borland C++ compiler comes with a profiler known as Turbo Profiler, or TPROF for short. Before you purchase a more powerful profiler, give this one a whirl. My own experience is that the Borland profiler is a powerful tool, but one that sometimes falls short when asked to profile very large programs or code that uses floating point arithmetic. (TPROF has some problems determining how much of a program's time is spent executing floating point instructions, which is unfortunate when you consider that floating point can be one of the major drags on program execution speed.)

More About Fixed Point Math

Now that we know *where* to optimize a program, let's talk a little more about *how* to optimize it. In particular, let's look into some of the details of fixed point math.

To understand how fixed point math works, think about how numbers with fractional values are written using the decimal numbering system. The number that represents one plus half of one, for instance, is written as 1.5. The number to the left of the decimal represents the integral (that is, whole number) portion of the number, while the number to the right of the decimal represents the fractional portion of the number. Here are a few more decimal fractions:

```
152.234
-7.3333
890234.98017203322
```

In each case, the numbers to the left of the decimal point represent integral amounts and the numbers to the right are fractional amounts.

We can write binary numbers that have fractional values in the same way:

```
1100011101.11110001001
1.01
-1110.0000001
```

The rules for binary fractions are the same as for decimal fractions. The digits to the left of the decimal point represent integral values and the digits to the right are the fractions. (Strictly speaking, binary numbers don't have "decimal points" at all, since they aren't decimal numbers. However, it's easier to use the familiar term.)

An interesting property of numbers with decimal points is that when we perform certain mathematical operations on them — in particular, multiplication and division — the digits in the result are the same no matter where the decimal points in the original numbers were located. Only the position of the decimal changes. For instance, suppose we want to multiply the two decimal numbers 1.3 and 41.7:

```
1.3 x 41.7 = 54.21
```

The result is 54.21. But suppose we move the decimal point in the first number one position to the right and multiply instead the numbers 13 and 41.7:

```
13 x 41.7 = 542.1
```

Interestingly, the only difference in the result is that the decimal point also moves one position to the right. What will happen if we also move the decimal point in the second number one position to the right and multiply the decimal numbers 13 and 417? This is what will happen:

```
13 x 417 = 5421
```

Now the decimal point has moved *two* spaces to the right.

The reason for this becomes apparent if you recall what you were taught in elementary school about "long multiplication." In a multiplication such as

```
  41.7
x 1.3
------
 54.21
```

the number of decimal positions to the right of the decimal point in the result is the *sum* of the decimal positions to the right of the decimal point in the multiplier and multiplicand (the two numbers being multiplied). This rule also applies to binary multiplication — and it's the secret of fixed point multiplication. It doesn't matter where we place the decimal point in a binary number as long as (a) we remember where we put it; and (b) we understand where it will be after we perform a mathematical operation on that number.

For instance, if we multiply a binary number that has two digits after the decimal point by a binary number that has three digits after the decimal point, the result will have five digits after the decimal point — but the digits themselves will be exactly the same as if there had been no decimal point at all. This means that we can perform fixed point math with the same integer math instructions that we use on numbers without decimal points, as long as we keep track of the position of the decimal point after each operation and treat the numbers accordingly.

Addition and subtraction are a great deal easier to perform on fixed point numbers, because they don't move the decimal point at all. In fact, the standard C/C++ and assembly language addition and subtraction instructions can be used with fixed point numbers. However, you must be sure that the decimal points are in the same positions in the numbers to be added or subtracted, even if it's necessary to shift the decimal point into the proper position before adding or subtracting. We'll see how to perform such a shift in a moment.

In the code that we develop in this chapter, we'll assume that all fixed point numbers are 32-bit *long* values with a decimal point right in the center, so that each fixed point number will have 16 digits to the left of the decimal point and 16 digits to the right of the decimal point. We'll refer to such numbers in this chapter as 16:16 numbers, where the colon represents the decimal point and the numbers to both sides of the colon represent the number of digits on those sides. There's no rule that says there *have* to be that many digits on both sides of the decimal point. You can place the decimal point anywhere that you find useful, as long as you remember where it is. For instance, if you are dealing strictly with numbers in the range 0 to 1, you might want to use 1:31 fixed point numbers, with only one digit to the left of the decimal point and the remaining 31 digits to

the right. This will give you much greater fractional precision. For our purposes, however, 16 binary digits of fractional precision will be enough.

Follow That Decimal!

Now that you know how to construct a fixed point number, the next thing you need to know is how to perform arithmetic on that number. Actually, you can use ordinary addition, subtraction, multiplication, and division operations between fixed point numbers (and between fixed point and regular numbers). But you must always be aware where the decimal point is after performing those operations and be prepared to move the decimal point elsewhere if necessary.

Following the decimal point in a fixed point number is a matter of learning a couple of rules. As you've seen, when multiplying two numbers, fixed point or otherwise, the number of digits after the decimal point in the result will be the sum of the number of digits after the decimal point in the two numbers being multiplied. When dividing two numbers, the number of digit positions after the decimal point in the result will be the number of digit positions in the dividend (the number being divided into) minus the number of digit positions in the divisor (the number being divided into it).

Since we'll be using 16:16 fixed point numbers in this book, each of which will have 16 digit positions after the decimal point, we know that after multiplying two of these numbers there will be 32 digits after the decimal point. But wait a minute! The largest integer type offered in C++ (or by the 80x86 CPU) is only 32 bits long. That means that the result of fixed point multiplication will leave us with *only* the digits to the right of the decimal point. The 16 digits to the left of the decimal point will simply vanish. Fortunately, if we write our fixed point multiplication code in 80386/486 assembly language, there's a trick that we can use to catch those digits and put them back where they belong before they're lost forever. (Of course, taking advantage of these instructions means that our fixed point routines will not work on 80286 or earlier processors. But few games developed after 1983 will run on the 286, so it's less than imperative for us to support it. If you want your game to run on a 286, you'll either have to avoid fixed point or write your own fixed point routines using 286, or C++, instructions.)

A similar problem occurs during fixed point division. The 16 digits to the right of the decimal point will be shifted into oblivion during a division operation between two 16:16 fixed point numbers, unless we take steps to preserve those digits. As you might guess, assembly language also provides us with a trick we can use to do this.

Both of these tricks are made possible by left- and right-shift instructions.

Shifting Left and Right

A shift instruction shifts the decimal point — and all of the digits — in a number to the right or left by a given number of digit positions. There are two types of shift instructions: left shifts and right shifts. These operations shift the digits and decimal to the left and right, respectively. (The directions left and right as used here are relative to the standard orientation of digits in a number, with the most significant digits — that is, those digits that represent the largest values — to the left and the least significant digits to the right.

Microcomputer shift operations only work on numbers written in the binary notation system, because that's the way in which the computer treats numbers on the CPU level. For instance, if we shifted all digits in the binary number 00100 two digits to the left, it would become the number 10000. Shifting those digits two positions to the right would produce the result 00001 (or just 1). See Figure 12-1 for an illustration.

The C/C++ right-shift operation is represented by the symbols ">>" and the left-shift operation is represented by the symbols "<<". The number to be shifted should be placed to the left of these operators and the number of digit positions by which the number is to be shifted is represented by the number to the right of these operators. To shift the digits in the number represented by the variable *a* two digits to the right, you would write:

```
a >> 2
```

Similarly, to shift the digits in *a* two digits to the left, you would write:

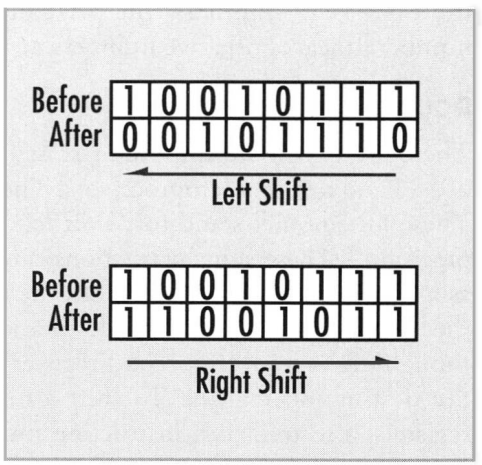


Figure 12-1 Left-shift and right-shift operations on binary numbers

```
a << 2
```

Note that this doesn't actually change the value of the variable *a,* which is still set to the original unshifted value. Instead, the shifted value of *a* becomes the value of the expression itself. To shift the digits in *a* three positions to the left and store that value back in *a,* you would write:

```
a <<= 3;
```

The 80x86 instruction set includes several shift instructions. The SHR and SHL instructions shift the digits in a CPU register or memory location by a specified number of digit positions to the right and left, respectively. The number of positions must be placed in the 8-bit CL register. (If it only needs to be shifted by one position, the number 1 may be used instead of the value in the CL register.) For instance, to shift the digits in the 16-bit DX register by seven positions to the right, you would write:

```
mov cl,7
shr dx,cl
```

Arithmetic Shifts

The SHL and SHR instructions are sometimes called logical shifts, because they shift *all* of the digits in a binary number, regardless of what those digits represent. The 80x86 instruction set also features a pair of arithmetic shift instructions: SAL and SAR. The SAL instruction is, in fact, identical to the SHL instruction. But the SAR instruction is slightly different from SHR, in that it doesn't affect the binary digit on the far left end of the number being shifted. Since this digit is sometimes used as a sign bit, to represent positive (if the digit is 0) or negative (if the digit is 1) numbers, the SAR instruction doesn't change the sign of the number. It leaves negative numbers negative and positive numbers positive.

Double-Precision Shifts

There are two more important shift instructions that are only available on the 80386 and newer microprocessors. These are the SHLD and SHRD instructions. These mnemonics stand for "shift left double-precision" and "shift right double-precision." These shift instructions can be used to treat two 32-bit CPU registers as a single 64-bit register and shift the digits across all 64 bits. Oddly, only one of the two registers (called the destination register) is changed by this operation, though digits from the second register (called the source register) are shifted into the destination register. To shift a 64-bit number in, say, the EAX and EDX registers, a second shift instruction must be used to shift the source register, like this:

```
mov  cl,24
```

```
shld edx,eax,cl
shl  eax,cl
```

This shifts the 64-bit number in the EAX and EDX registers by 24 digit positions. We can also use a 32-bit memory location as the destination register, like this:

```
shld [memory_address],eax,cl
```

A Fixed Point Multiplication Function

By using these shift instructions, along with the standard 80x86 assembly language multiplication and division instructions, we can create a set of assembly language functions for multiplying and dividing fixed point numbers. (The code that follows is adapted from a set of fixed point routines developed by game programmer Kevin Gliner, who generously shared his ray-casting code with me for this book.) We'll make these functions callable from C++, but they can easily be adapted as necessary for use in other programming languages.

The fixed point function that we'll use most often in this chapter is the one for multiplication, so we'll develop that one first. We'll declare the function as a PROC called *_fixmul:*

```
_fixmul        PROC
```

We'll pass two parameters to this function, representing the two numbers to be multiplied. We'll call them ARG1 and ARG2:

```
ARG arg1:DWORD, arg2:DWORD
```

As usual, we need to set up the BP register to point to the CPU stack, if we want to be able to read the parameters that we've passed to the function:

```
push bp
mov  bp,sp
```

We'll put ARG1 into the 32-bit EAX register:

```
mov  eax,[arg1]
```

The actual fixed point multiplication is simple. We can use the 80x86 IMUL (for "integer multiplication") to multiply EAX by ARG2:

```
imul arg2
```

However, as noted earlier, the multiplication of a pair of 16:16 fixed point numbers will leave 32 digits to the right of the decimal point in the result. Indeed, after we execute the previous instructions, those 32 digits will be sitting in the EAX register. What happens to the digits to the *left* of the decimal point? Fortunately, the 80x86 takes care of them: They've been carefully placed in the

EDX register. Our 16:16 fixed point number has now become a 32:32 fixed point number spanning two registers.

Before we pass the result of this multiplication back to C++, we'll need to convert it back to a 16:16 fixed point number. To fit the conventions of the C++ compiler for passing 32-bit results from functions, the low order 16 bits of this number must go in AX and the high order 16 bits in DX. Since the AX and DX registers are simply the low order 16 bits of the EAX and EDX registers, this job is already half done, because the low order 16 bits of the 16:16 number were left in the high order 16 bits of registers EDX by the multiplication. We must move these digits to the lower 16 bits of the register. One way to do this is to use

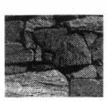
```
shrd  eax,edx,16
```

This shifts the high 16 bits of EAX into the low 16 bits of EAX, without affecting EDX. Why don't we simply use an SHL or SAL instruction on EAX? Because we'd need to load the number of bit positions to shift into the CL register first, which would slow down the operation of the function. Since the number in DX:AX will be automatically passed back to C++, all we need to do is restore the value in the BP register, return to the calling routine, and end the PROC.

```
pop  bp
ret
_fixmul  ENDP
```

We can call this function from C++ by using the name *fixmul()*. You'll notice that the underscore at the beginning of the name should be omitted when calling the function from C++. The workings of the multiplication and shift instructions are shown in Figure 12-2.

The complete text of the *fixmul()* function appears in Listing 12-1.



Listing 12-1 The fixmul() function

```
_fixmul        PROC
  ARG arg1:DWORD, arg2:DWORD
  push bp            ; Set up BP register.
  mov  bp,sp
  mov  eax,arg1      ; Get first argument into EAX
  imul arg2               ; Multiply it by second argument
  shrd eax,edx,16         ; Shift high and low bytes into DX:AX
  pop  bp
  ret
_fixmul        ENDP
```

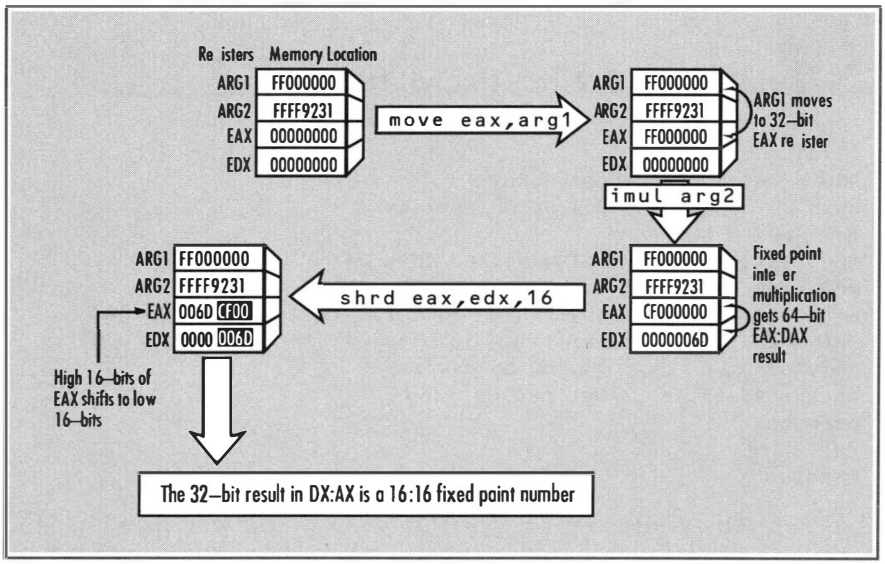


Figure 12-2 The contents of the registers before and after the multiplication and shift instructions

A Fixed Point Division Function

Using similar logic, we can create a fixed point division instruction that will divide a pair of 16:16 numbers and produce a 16:16 result. The main difference is that we must perform the shift *before* we perform the division, turning the 16:16 dividend into a 32:32 number before we divide it by the 16:16 divisor. The division produces a 32-bit result, with 16:16 precision, just the way we want it. We'll use the 80x86 IDIV instruction, which divides a 32-bit divisor into a 64-bit dividend in the EDX and EAX registers, and leaves a 32-bit result in EAX. The text of the *fixdiv()* function is in Listing 12-2.

Listing 12-2 The fixdiv() function

```
_fixdiv       PROC
 ARG numer:DWORD,denom:DWORD
 push bp          ; Set up BP register.
 mov  bp,sp
 mov  eax, numer  ; Put dividend into EAX
 mov  edx,eax     ; Copy it into EDX
 sar  edx,16      ; Shift high 16 bits of EDX back into EAX
 shl  eax,16      ; Shift low 16 bits of EAX into high 16 bits
 idiv denom       ; Divide by divisor
 shld edx,eax,16  ; Get result
 pop  bp
 ret
_fixdiv       ENDP
```

A potentially confusing part of this code is the way in which the shift is performed. To convert the 16:16 number in EAX into a 32:32 number in EDX:EAX, the value in EAX is first copied to EDX, then an SAR instruction is used to put the high 16 bits of that instruction into the low 16 bits of EDX, where they belong. This also serves to clear any earlier values out of the high 16 bits of EDX. (If the dividend is a positive number, this will fill the high 16 bits of EDX with 0s. If it's a negative number, this will fill the high 16 bits with 1s, which preserves the sign of the number and allows us to use this function with both negative and positive operands.) The SHL instruction then shifts the low 16 bits of EAX into the high 16 bits, filling the low 16 bits with 0s. Thus the 16:16 number becomes a 32:32 number. After the division, the SHLD instruction gets the 16:16 result into the proper registers to be passed back to C++.

Fixed Point Trigonometry

Even when working in fixed point, we'll need to perform such trigonometric functions as sine and cosine. The Borland C++ libraries don't contain any fixed point trigonometric functions, but it's quite easy to use the floating point trig routines in the Borland libraries to create a table of fixed point sines and cosines, which can then be used in our program with the table look-up technique we discussed earlier in this chapter.

We'll create a short program called MAKETRIG.CPP that will generate an ASCII text file which will contain the initialization statements for a pair of fixed point (that is, long-integer) arrays called *sin_array* and *cos_array*. This file will be called TRIG.CPP. MAKETRIG and will also generate the header file TRIG.H,

which will contain definitions for macros that can be used to perform trigonometric functions using the tables in TRIG.H.

MAKETRIG.CPP will begin with some constant definitions:

```
const NUMBER_OF_DEGREES = 4096;          // Degrees in a circle
const SHIFT = 16;                        // Fixed point shift
const long SHIFT_MULT = (1L<<SHIFT);     // Fixed point shift as
                                         //  a multiplication
```

The constant NUMBER_OF_DEGREES is set equal to 4096 because that's the number of "degrees" that we'll be using in our fixed point polar coordinate system. This will allow us to have a fair amount of precision in the rotation of rays and objects while not having to spend time interpolating fractions of degrees. In a moment, I'll show you one method of converting the radian-based sines and cosines supported by the BC++ libraries into our 4096-degree system.

The SHIFT constant represents the number of degrees that the decimal point in our fixed point numbers has been shifted left relative to the decimal point in a standard integer (which is all the way down at the righthand end of the number). We can use this constant to translate standard integers into fixed point numbers. For instance, if we wanted to translate the value of the *int* variable *int_number* into a fixed point number in the *long* variable *fix_number*, we would write:

```
fix_number = (long)int_number << SHIFT;
```

Notice that it's necessary to cast the value of *int_number* to a *long* value before performing the shift. Although this conversion will occur automatically when the value of the expression is assigned to the *long* variable *fix_number*, the shift operation will already have destroyed the digits in *int_number* by shifting them off the end of what it thinks is a 16-bit value. The typecast tells the shift to treat *int_number* as a 32-bit value, rescuing the shifted digits.

The use of a constant here for the SHIFT value will allow us to change more easily to other precisions of fixed point numbers, if that should become necessary or desirable, though changes will still need to be made to the assembly language functions that deal with fixed point numbers.

Finally, the SHIFT_MULT constant is used to convert floating point numbers, which cannot be shifted, into fixed point numbers using a simple multiplication. For instance, the value of the *float* variable *float_number* can be converted to a fixed point value in the long integer variable *fix_number* like this:

```
fix_number = float_number * SHIFT_MULT;
```

(Note that we can't cast *float_number* into a *long* and then perform a shift on it because we'd lose the fractional portion of its value, which is precisely what our fixed point math system is designed to preserve.)

The MAKETRIG program will be fairly short, consisting of only a single *main()* function:

```
void main()
{
```

We'll need a variable to keep track of radians, as we translate from radians to our custom degree system. Time not being of the essence here, we'll make this variable a *double,* to guarantee a high level of accuracy:

```
double radians=0.0;
```

We'll also need a file handle for the TRIG.H and TRIG.CPP files. We'll give that handle the unimaginative name fname:

```
FILE  *fname;
```

Finally, we'll need an array of char for translating our fixed point numbers into ASCII characters for output, a task that's more difficult than it might at first appear:

```
char hex[20];
```

We'll create the file TRIG.H first:

```
fname=fopen("trig.h","wt");
```

This opens the file TRIG.H as a write-only text file. Now we'll write a pair of macro definitions to the file:

```
fprintf(fname,"#define COS(X) cos_table[X&(NUMBER_OF_DEGREES-1)]\n");
fprintf(fname,"#define SIN(X) sin_table[X&(NUMBER_OF_DEGREES-1)]\n");
```

In C and C++, a macro is a kind of constant that represents a string of characters and can take one or more parameters, almost like a function. These two macros perform the sine and cosine functions using table look-ups. Each macro takes a single parameter, X, which represents the angle for which we want the sine or cosine. The string of characters that we pass to the macro will then replace the character X in the macro text. In each case, the value that we give X will be ANDed with the NUMBER_OF_DEGREES minus 1. When NUMBER_OF_DEGREES is a power of two (as it always should be), this will give us the value of x modulo the number of degrees (that is, the remainder after X has been divided by the number of degrees). The resulting value is then used as an index into the sine or cosine table.

(Why use macros instead of functions to perform these operations? The compiled version of a standard C/C++ function contains a lot of code that will slow these operations down. For instance, the function will be called with the assembly language CALL instruction and terminated with a RET, with additional

code to set up the stack. A macro avoids this overhead. Alternatively, we could use what is known in C++ as an *inline function*, which works much like a macro.)

We also want to print our constant definitions in the TRIG.H file, so that they'll be accessible to our programs:

```
fprintf(fname,"\nconst NUMBER_OF_DEGREES = %d;\n",
     NUMBER_OF_DEGREES);
fprintf(fname,"const SHIFT = %d;\n",SHIFT);
fprintf(fname,"const SHIFT_MULT = 1<<SHIFT;\n\n");
```

These instructions will simply duplicate in the TRIG.H file the same constant definitions that we've put at the head of MAKETRIG.CPP.

We'll be storing the trig tables in the arrays cos_table and sin_table, so we'll add extern definitions for these arrays to TRIG.H:

```
fprintf(fname,"extern long cos_table[%d];\n",
     NUMBER_OF_DEGREES);
fprintf(fname,"extern long sin_table[%d];\n",
     NUMBER_OF_DEGREES);
```

And that's all that we need in TRIG.H, so we'll close the file:

```
fclose(fname);
```

Now let's build the second file, TRIG.CPP:

```
fname=fopen("trig.cpp","wt");
```

For clarity's sake, let's open TRIG.CPP with some contents identifying the contents of the file:

```
fprintf(fname,"\n//FIX.CPP\n");
fprintf(fname,"//  Fixed point math tables\n\n");
```

The rest of the file will be nothing but the initialization statements for the two arrays. We'll start with cos_table:

```
fprintf(fname,"long cos_table[%d]={\n    ",
     NUMBER_OF_DEGREES);
```

That will declare cos_table as an array of *longs* with NUMBER_OF_DEGREES elements. The equal sign and the left curly bracket indicate that this will be followed by a list of 4096 fixed point cosine values. We'll calculate those values in a for() loop that iterates 4096 times. First, however, we'll create an *int* variable called *count* to help us format the data properly:

```
int count=0;
```

More about the *count* variable in a moment.

Now we can begin the for() loop:

```
or (int degree=0; degree<NUMBER_OF_DEGREES; degree++) {
```

The loop variable *degree* represents the number of degrees for which we wish to calculate the cosine, in our 4096-degree system. Before we can use the standard BC++ trigonometric functions to perform the calculation, though, we must convert the number of degrees into radians. For this purpose, we'll create a *double* variable called *radians*. To convert degrees into radians, we'll divide the current number of degrees (which we'll first cast to a double) by the NUMBER_OF_DEGREES, then multiply the result by the number of radians in a circle, which is 2 * PI:

```
double radians = (double)i/NUMBER_OF_DEGREES*3.14159*2;
```

We'll use the standard *cos()* function to calculate the cosine of *radians*, then convert it to fixed point using SHIFT_MULT and store the resulting value in the temporary *long* variable *temp_long*:

```
long temp_long=(long)(cos(radians)*SHIFT_MULT);
```

Now we have to translate this value to ASCII to output it to the file. Ordinarily, we would do this using the formatting capabilities of the fprintf statement, but these capabilities have more than a little trouble with outputting long integers, especially negative long integers. Instead, we'll use the standard *ltoa()* function, which converts long integers into strings of ASCII characters:

```
ltoa(temp_long,hex,16);
```

This converts the value of *temp_long* to a hexadecimal ASCII string and stores the result in the char array hex, which we declared earlier. Now we can use *fprintf* to output this string of characters to the file FIX.CPP, with a leading "0x" to indicate that the string represents a hexadecimal number:

```
fprintf(fname,"0x%s, ",hex);
```

So that our 4096-element initialization statement doesn't become one long line with no carriage returns, we'll output a return after every eighth element. The variable *count* will be incremented by one on each pass through the loop, so that we'll know when eight elements have been printed:

```
count++;
```

When *count* reaches eight, we print a carriage return, plus a four-character indentation at the beginning of the next line:

```
if (count>=8) {
  fprintf(fname,"\n    ");
```

And we reset *count* to 0:

```
  count=0;
}
```

When all 4096 elements have been printed, we'll output a closing curly bracket, plus two carriage returns:

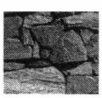
```
}
fprintf(fname,"};\n\n");
```

The code for the sine table is almost identical to this, except that the trigonometric function has been changed:

```
fprintf(fname,"long sin_table[%d]={\n    ",
     NUMBER_OF_DEGREES);
count=0;
for (i=0; i<NUMBER_OF_DEGREES; i++) {
     radians = (double)i/NUMBER_OF_DEGREES*3.14159*2;
     long temp_long=(long)(sin(radians)*SHIFT_MULT);
     ltoa(temp_long,hex,16);
     fprintf(fname,"0x%s, ",hex);
     count++;
     if (count>=8) {
          fprintf(fname,"\n    ");
          count=0;
     }
}
fprintf(fname,"};\n");
```

The MAKETRIG.CPP Module

The complete text of MAKETRIG.CPP appears in Listing 12-3. The text of the TRIG.H file that it generates appears in Listing 12-4. The TRIG.CPP file is much too long to reprint here, but a small portion of both the sin_table[] and cos_table[] arrays are shown in Listing 12-5. The complete file can be found on the disk in the OPTIMIZE directory, along with the other program files in this chapter.



Listing 12-3 The MARKETRIG.CCP program

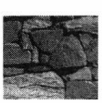
```
#include  <stdio.h>
#include  <stdlib.h>
#include  <math.h>
#include  <conio.h>
#include  <string.h>

const NUMBER_OF_DEGREES = 4096; // Degrees in a circle
const SHIFT = 16;               // Fixed point shift
const long SHIFT_MULT = (1L<<SHIFT);   // Fixed point shift as
                                //  a multiplication
```

continued on next page

continued from previous page

```
void main()
{
  double radians=0.0;
  FILE  *fname;
  char hex[20];

  // Create file TRIG.H for constants and macros:
  fname=fopen("trig.h","wt");

  // Macro definitions for trig functions:
  fprintf(fname,"#define COS(X) cos_table[X&(NUMBER_OF_DEGREES-1)]\n");
  fprintf(fname,"#define SIN(X) sin_table[X&(NUMBER_OF_DEGREES-1)]\n");

  // Constant definitions:
  fprintf(fname,"\nconst NUMBER_OF_DEGREES = %d;\n",
          NUMBER_OF_DEGREES);
  fprintf(fname,"const SHIFT = %d;\n",SHIFT);
  fprintf(fname,"const SHIFT_MULT = 1<<SHIFT;\n\n");

  // External table declarations:
  fprintf(fname,"extern long cos_table[%d];\n",
          NUMBER_OF_DEGREES);
  fprintf(fname,"extern long sin_table[%d];\n",
          NUMBER_OF_DEGREES);
  fclose(fname);

  // Create file TRIG.CPP for data tables:
  fname=fopen("trig.cpp","wt");
  fprintf(fname,"\n//TRIG.CPP\n");
  fprintf(fname,"//  Fixed point math tables\n\n");

  // Create cosine table:
  fprintf(fname,"long cos_table[%d]={\n    ",
          NUMBER_OF_DEGREES);
  int count=0;

  // Loop through 4096 degrees:
  for (int i=0; i<NUMBER_OF_DEGREES; i++) {

    // Translate degrees into radians:
    radians = (double)i/NUMBER_OF_DEGREES*3.14159*2;

    // Get cosine:
    long temp_long=(long)(cos(radians)*SHIFT_MULT);

    // Convert cosine to hexadecimal ASCII:
    ltoa(temp_long,hex,16);

    // Print it to file:
     fprintf(fname,"0x%s, ",hex);

    // Output carriage return if eighth entry printed:
```

```
      count++;
      if (count>=8) {
        fprintf(fname,"\n    ");
        count=0;
      }
    }
    fprintf(fname,"};\n\n");

    // Create sine table:
    radians=0.0;
    fprintf(fname,"long sin_table[%d]={\n    ",
            NUMBER_OF_DEGREES);
    count=0;

    // Loop through 4096 degrees:
    for (i=0; i<NUMBER_OF_DEGREES; i++) {

      // Translate degrees into radians:
      radians = (double)i/NUMBER_OF_DEGREES*3.14159*2;

      // Get cosine:
      long temp_long=(long)(sin(radians)*SHIFT_MULT);

      // Convert cosine into hexadecimal ASCII:
      ltoa(temp_long,hex,16);

      // Print it to file:
       fprintf(fname,"0x%s, ",hex);

      // Output carriage return if eighth entry printed:
       count++;
      if (count>=8) {
        fprintf(fname,"\n    ");
        count=0;
      }
       }
    fprintf(fname,"};\n");

    // Close file:
    fclose(fname);
}
```



Listing 12-4 The TRIG.H file

```
#define COS(X) cos_table[X&(NUMBER_OF_DEGREES-1)]
#define SIN(X) sin_table[X&(NUMBER_OF_DEGREES-1)]

const NUMBER_OF_DEGREES = 4096;
const SHIFT = 16;
```

continued on next page

continued from previous page

```
const SHIFT_MULT = 1<<SHIFT;

extern long cos_table[4096];
extern long sin_table[4096];
```

Listing 12-5 Portions of the TRIG.CPP module

```
//TRIG.CPP
//Fixed point math tables
long cos_table[4096]={
    0x10000, 0xffff, 0xffff, 0xffff, 0xfffe, 0xfffe, 0xfffd, 0xfffc,
    0xfffb, 0xfff9, 0xfff8, 0xfff6, 0xfff4, 0xfff2, 0xfff0, 0xffee,
    0xffec, 0xffe9, 0xffe7, 0xffe4, 0xffe1, 0xffdd, 0xffda, 0xffd7,
    0xffd3, 0xffcf, 0xffcb, 0xffc7, 0xffc3, 0xffbf, 0xffba, 0xffb5,
....
long sin_table[4096]={
    0x0, 0x64, 0xc9, 0x12d, 0x192, 0x1f6, 0x25b, 0x2bf,
    0x324, 0x388, 0x3ed, 0x451, 0x4b6, 0x51a, 0x57f, 0x5e3,
    0x648, 0x6ac, 0x711, 0x775, 0x7da, 0x83e, 0x8a3, 0x907,
    0x96c, 0x9d0, 0xa35, 0xa99, 0xafd, 0xb62, 0xbc6, 0xc2b,
    0xc8f, 0xcf4, 0xd58, 0xdbc, 0xe21, 0xe85, 0xeea, 0xf4e,
....
```

Optimized Wall Drawing

Now that we have some grounding in the basic principles of optimization, let's start optimizing code. We'll put all of our optimization into the *draw_maze()* function of our ray-casting engine, because this is where our programs will be spending the majority of their time. As noted earlier, we'll rewrite three portions of the *draw_maze()* function in machine language, as separate assembly language PROCs. These PROCs will replace the portions of the code that draw the walls, the floors, and the ceilings. The code in these three PROCs is adapted from code written by Kevin Gliner for his own ray-casting engine. Any bugs or bottlenecks in the code were added by me in the translation.

We'll restrict the wall-drawing optimization to the loop that copies a column of a bitmap to the wall, stretching or shrinking it as necessary by adding or subtracting pixels. We'll continue performing the rest of the wallcasting in C++, though we'll use fixed point math in doing so, as you'll see in a moment. To get things under way, let's write a function called *drawwall():*

```
_drawwall PROC
```

We'll use a slightly different method of drawing the wall in this assembly language function than we used earlier in our C++ ray caster. Before, we used an incremental division method similar to that used in Bresenham's algorithm to decide when a pixel is to be repeated or skipped. Now, we'll use a simpler method that wasn't feasible without the aid of fixed point math. This method involves dividing the size of the wall column to be drawn into the size of the column of pixels to be mapped onto it to produce a value that we'll call the *increment*. The increment is a value which can have a fractional portion and which is added to the current pixel offset into the bitmap to determine which pixel is to be drawn next. We'll truncate the fractional portion before indexing into the bitmap, but we'll retain the fractional portion when we add the increment to the offset. Thus if the value of the increment is, say, 0.5, then the bitmap offset will be advanced by one pixel on every other iteration of the drawing loop, since 0.5 must be added to the offset twice to move it forward by 1. If the value of the increment is 1.5, then the pointer will be advanced by 1 pixel, then 2, then 1 again — and so forth.

We'll also make a minor change in the way the bitmaps are stored. If you look at the file Walls.pcx in the OPTIMIZE directory, you'll see that the tiles are now lying on their sides, so that each "column" of each bitmap is now a horizontal row. This will make it easier to move along the columns, as you'll see. And we'll now be drawing the walls from bottom to top instead of from top to bottom, which is why the bitmaps have "fallen over" onto their right sides rather than their left sides.

The *drawwall()* function will accept five parameters from the calling function. These are a far pointer to the address in the video display or screen buffer where the bottommost pixel in the wall column is to be drawn (*screenptr*); a far pointer to the first pixel to be drawn from the bitmap (*bitmapptr*); an integer representing the height in pixels of the visible wall column to be drawn (*height*); a 16:16 fixed point number representing the increment to be added to the pixel offset after each visible pixel is drawn (*increment*); and a far pointer to the position in the lightsourcing table at which the 256 light levels can be found for the wall's light level (*litelevel*):

```
ARG screenptr:DWORD,bitmapptr:DWORD,height:WORD,increment:
DWORD,litelevel:DWORD
```

We're going to completely unroll the main loop of this function, the loop that actually puts the pixels in the screen buffer. You might wonder how we're going to know in advance how many times to unroll the loop, since the number of iterations will depend on the height of the visible wall column. In fact, we're going to unroll it a full 200 times, which is the maximum height that the visible

wall column can have on the mode 13h display. If we need to draw fewer pixels — and we usually will — we'll jump into the middle of this unrolled loop, effectively skipping over the pixels that don't need to be drawn. To calculate how many pixels we must skip, we'll take the *height* value passed to us from the calling routine and subtract it from 200, like this:

```
mov   bx,height
mov   ax,200
sub   ax,bx
```

The number of pixels to be skipped is now in AX. We'll get back to it in a moment. In the meantime, we'll store important values in registers. For instance, we'll store *increment* in ECX:

```
mov   ecx,increment
```

And we'll store *screenptr*, *bitmapptr*, and *litelevel* in gs:si, fs:di, and es:bx, respectively:

```
lgs   si,screenptr
lfs   di,bitmapptr
les   bx,litelevel
```

We need to be sure that the EBX register has nothing but 0s in it. We'll do this in the ordinary straightforward fashion, by moving a 0 into it:

```
mov   ebx,0
```

We'll also store a constant value of 320 — the width of the display in pixels — in the BP register for later use:

```
mov   bp,320
```

Now let's calculate where to jump into the loop. Each iteration of our unrolled loop will be 21 bytes long, so we'll multiply the number of pixels to skip (which we left in AX) by 21:

```
imul  ax,21
```

Then we'll add the starting address of the loop, which will be at address walloop, to this value:

```
mov   di,offset walloop
add   di,ax
```

The address to which we need to jump is now in DI, so we'll clear out EAX:

```
xor   eax,eax
```

And we'll jump to the address in DI:

```
jmp   di
```

We'll create the loop itself as a macro, so we don't have to type it 200 times. A macro is defined using the MACRO directive. We'll call this macro columnloop:

```
COLUMNLOOP MACRO
```

Before we can put a pixel on the display, we need to index into the bitmap to find the pixel color. The offset for the pixel is in EDX, but it's a 16:16 fixed point number, which won't work properly as an offset. This can be easily fixed by using an SHLD instruction to move the high 16 bits of EDX, which contain the integral portion of the offset, into the low 16 bits of the EDI register:

```
shld edi,edx,16
```

The DI register (which is the lower half of EDI) now contains the offset. We can use fs:di to fetch the pixel color into the AL register:

```
mov  al,fs:[di]
```

We want to place the lightsourced equivalent of this value onto the display. We've put a pointer to the lightsource table in es:bx and we can use the color itself, in AX, to index into the table:

```
mov  al,es:[ebx + eax]
```

We have to use the full 32-bit registers for this particular addressing mode, which is why we cleared both EBX and EAX earlier, to be certain there were no stray digits in the upper 16 bits of either.

The pixel color is now in AL. Since gs:si points at the screen or screen buffer location for the next pixel to be drawn, we simply copy the value from AL to the address pointed to by those registers:

```
mov  gs:[si],al        ;mov pixel color into ScreenBuffer
```

We then subtract the increment from the pointer into the bitmap:

```
sub  edx,ecx
```

And we subtract 320 (which we placed earlier in BP) from the screen pointer in SI:

```
sub  si,bp
```

In both of these cases, we subtract rather than add because we're drawing the screen column from the bottom to the top.

That's the end of the macro. Back in the *drawwall()* function we repeat the macro 200 times like this:

```
walloop:
    REPT 200
      COLUMNLOOP
    ENDM
```

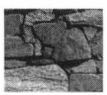
That sure beats typing the contents of the loop 200 times!

All that we need to do now is terminate the PROC in the standard fashion:

```
  pop           bp
  ret
_drawwall  ENDP
```

The drawwall() Function

The complete text of the *drawwall()* function appears in Listing 12-6. The text of the COLUMNLOOP macro is in Listing 12-7.



Listing 12-6 The drawwall() function

```
_drawwall PROC
          ARG
screenptr:DWORD,bitmapptr:DWORD,height:WORD,increment:DWORD,litelevel:DWO
RD
    push  bp              ; Save BP
    mov   bp,sp           ; Set up stack pointer
    mov   bx,height       ; Get height in BX
    mov   ax,200          ; Calculate number of pixels to skip
    sub   ax,bx           ; Leave result in AX
    mov   ecx,increment   ; Get increment in ECX
    lgs   si,screenptr    ; Get screen index in GS:SI
    lfs   di,bitmapptr    ; Get pointer to bitmap in FS:DI
    mov   ebx,0           ; Clear out EBX
    les   bx,litelevel    ; Get lightsource table addr. in BX
    mov   dx,di         ; Copy increment in DX
    shl   edx,16          ; Reverse the bytes
    imul  ax,21           ; Calculate jump address
    mov   di,offset walloop ; Add start of loop....
    add   di,ax           ; ...to offset in loop
    mov   bp,320          ; Store constant in BP
    xor   eax,eax         ; Clear out EAX
    jmp   di              ; Jump into unrolled loop.
walloop:
    REPT 200              ; Repeat macro 200 times
      COLUMNLOOP
    ENDM
    pop   bp        ; Restore BP
    ret
_drawwall   ENDP
```

Listing 12-7 The COLUMNLOOP macro

```
COLUMNLOOP MACRO
    shld edi,edx,16        ; Move integral portion of bitmap
                           ; pointer into DI
    mov  al,fs:[di]        ; Get color of next pixel
    mov  al,es:[ebx + eax] ; Get lightsourced color
    mov  gs:[si],al        ; Copy pixel color to screen column
    sub  edx,ecx           ; Add increment to bitmap pointer
    sub  si,bp             ; Point to next pixel in wall column
ENDM
```

Optimized Floors and Ceilings

The optimized floor and ceiling drawing routines will use a somewhat different algorithm than we used in our C++ ray-casting function. Instead of drawing the floors and ceilings column by column, starting at the tops and bottoms of walls and drawing to the tops and bottoms of the viewport, we'll draw the floors and ceilings in rows from left to right, checking to see if there's a wall intersecting each column before we draw a pixel in it.

Why are we using this method? What advantage does it give us over the algorithm we used earlier?

You'll recall that in our earlier floorcasting and ceilingcasting routines, it was necessary to rotate a ray for every pixel in the floor and ceiling areas, to find which pixel within the fine coordinate system of the maze grid was intersected by that ray. But that's inefficient because it requires that trigonometric functions and multiplications be performed for each pixel, which slows the process considerably. Even with our new table-derived trig functions, there are still a couple of multiplications involved per pixel, which is a couple of multiplications too many. The floor and ceiling drawing loops will be executed, collectively, nearly 64,000 times when creating a ray-cast image in a full-screen viewport. Every CPU cycle that we can shave off of these loops will noticeably speed up our code.

Fortunately, there's a better way to do this. If we draw the floors and ceilings in horizontal rows, we need only rotate a ray through the leftmost and rightmost pixels in each row, to determine which floor pixels they strike. The rays that pass through the intermediate pixels in a column will all fall along a straight line connecting these two end pixels. All we need to do to find the floor pixels corresponding to these intermediate screen pixels is to determine the distance between the two end floor pixels, divide it by the number of intermediate pixels to find the increment between each pixel, then repeatedly add that increment to

the floor pixel position to walk our way from one side of the viewport to the other.

This concept is illustrated in Figure 12-3. At the bottom is the viewer, looking at the video display, as seen from above. Two rays shoot out from the viewer's eyes, one (labeled A) passing through the leftmost pixel of a floor column on the video display, the other (labeled B) passing through the rightmost pixel of that same floor column. The floor pixel struck by A will be the leftmost floor pixel visible in that row, and the floor pixel struck by B will be the rightmost floor pixel visible in that row. The dotted line from the leftmost floor pixel to the rightmost represents the floor pixels that will appear in the intermediate pixels on that row.

Suppose the leftmost floor pixel is at fine coordinates 1007,659 and the rightmost is at coordinates 911,980. We can find out how long the line between these two floor pixels is — that is, how many floor pixels that line will pass over in the x and y directions — by subtracting the second set of coordinates from the first. Thus we know that the line passes over 911 minus 1007, or -96, pixels in the x direction and 980 minus 659, or 321 pixels in the y direction. Now let's suppose that the viewport in which these two floor pixels represent the extremes is 200 pixels wide. The distance from the leftmost pixel to the rightmost pixel is therefore 199 pixels. That tells us that the distance between two floor points represented by adjacent screen pixels is -96 divided by 199, or -0.4824, pixels in the x direction and 321 divided by 199, or 1.613, pixels in the y direction.

This gives us a method for calculating the positions at which the rays cast through the intermediate pixels in that row strike the floor. We rotate a ray

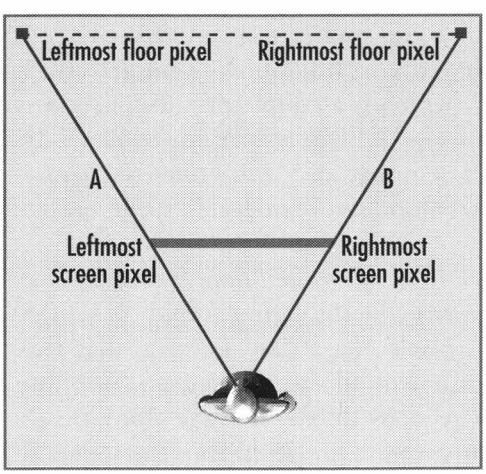


Figure 12-3 Calculating pixel positions along a line from left to right

through the leftmost and rightmost pixels in the row to determine which floor pixels those rays strike, then subtract the coordinates of the rightmost floor pixel from the coordinates of the leftmost floor pixel to get the distance between them and divide that distance by the distance between the two screen pixels to get the single pixel increment in each direction. We then add these increments repeatedly to the coordinates of the leftmost floor pixel to derive the coordinates of the intermediate pixels.

For instance, in the example above, we know that the leftmost floor pixel is at coordinates 1007,659 and the single pixel increment along the line connecting this pixel to the rightmost pixel is -0.4824,1.613. To get the coordinates of the floor pixel to be drawn on the screen immediately to the right of this one, we merely add these increments to the coordinates of the leftmost pixel. This will tell us that the coordinates of the second-to-the-rightmost pixel are 1006.5176,660.613. And the coordinates of the pixel to the left of that pixel are 1006.0352,661.226. If we keep adding these increments to the pixel coordinates for all of the pixels in a single row of the viewport, we can calculate the position of each without any multiplications or trigonometric table look-ups — except for those required to rotate the ray through the leftmost and rightmost rays.

This is the algorithm that we'll use when we translate a portion of the floor and ceiling drawing routines into assemby language. The portion of the floor and ceiling drawing routines that we're going to translate is the part that draws a single column of pixels. We'll call the resulting function *drawfloorrow()* and *drawceilrow()*. We'll look at *drawfloorrow()* in detail, since the two functions are nearly identical.

As before, we'll declare *drawfloorrow()* as an assembly language PROC:

```
_drawfloorrow  PROC
```

We'll be passing quite a few parameters to this function:

- *row* — the number of the viewport row in which we'll be drawing floor pixels
- *screenptr* — a far pointer to the position on the video display at which the row begins
- *texturelist* — a far pointer to an array of pointers to the upper-left corners of the 15 texture map tiles
- *floormap* — a far pointer to the 16x16 map of the floor
- *litelevel* — a far pointer to the 256-byte segment of the lightsourcing table for the distance at which the rays in the current column strike the floor

- *bots* — a far pointer to an array with an element for each column of the viewport containing the number of the row where the bottom of bottommost pixel in the wall appears in that column
- *xinc* — a fixed point number representing the increment in the x direction between floor pixels in the row
- *yinc* — a fixed point number representing the increment in the y direction between floor pixels in the row
- *x* — a fixed point number representing the x coordinate of the floor pixel visible through the leftmost screen pixel
- *y* — a fixed point number representing the y coordinate of the floor pixel visible through the leftmost screen pixel
- *w* — the width of the viewport

Here's the argument list from the program:

```
ARG  row:WORD,screenptr:DWORD,texturelist:DWORD,floormap:DWORD,
litelevel:DWORD,bots:DWORD,xinc:DWORD,yinc:DWORD,x:DWORD,y:DWORD,w:WORD
```

I'll explain the details of some of the more esoteric of these parameters as we go.

We'll need to use a lot of CPU registers in this function, so many that we won't be able to use BP to access the parameters after the main loop begins. So we'll move all of the parameters into either registers or memory locations, where they can be retrieved later:

```
mov   bx,w
mov   [width],bx
mov   ebx,litelevel
mov   [lightindex],ebx
mov   bx,row
mov   [rownum],bx
mov   [colnum],0
mov   ecx,y
mov   edx,x
lfs   si,screenptr
mov   ebx,xinc
mov   [xincrement],ebx
mov   ebx,yinc
mov   [yincrement],ebx
mov   ebx,texturelist
mov   [texture],ebx
mov   ebx,bots
mov   [botptr],ebx
les   bp,floormap
xor   eax,eax           ; Clear out EAX
```

Don't worry about the details of the above. I'll show you where the variables have been stashed as we use them.

Now we're ready to dive into the main loop. The main loop here is considerably longer than in *drawwall()*, so we won't unroll it completely. Instead, we'll unroll it eight times. Nonetheless, we'll put it in a macro, to save typing. We'll call that macro FLOORLOOP:

```
FLOORLOOP MACRO REP
```

This macro takes one parameter, REP. We'll set this equal to the number of times the macro has been repeated in the unrolled loop. The first time we call it, REP will be 0. The second, it will be 1. And so forth, up to the eighth time, when it will be 7. I'll show you what this parameter does in a moment.

We also need to establish a local address label in the macro. A local address label is a label that exists only within the macro. It ceases to exist the moment the macro is over. This avoids conflicts that could result when the same label name appears in multiple macro calls:

```
LOCAL SKIPPIXEL
```

This label will be used a bit later in the macro.

Earlier, I mentioned a parameter that points to an array containing the bottom positions of every wall in every column of the viewport. We'll use this array to determine whether we should draw a floor pixel in any given column of the current row. We can find out by comparing the number of the current row, which has been saved in the memory location ROWNUM, with the byte value stored in the current entry in this array, which is pointed to by the far pointer in the memory location BOTPTR. We'll put BOTPTR in the gs:bx registers, load the bottom position for the current column into AL, get the number of the current row in BX, and compare the two. If the row number isn't greater than the bottom of the wall, we'll jump over most of the code that follows, to the address represented by the local label SKIPPIXEL. This will circumvent *all* of the pixel drawing code for this column:

```
    lgs  bx,[botptr]  ; Get pointer to BOTS array
mov  al,gs:[bx]   ; Get current bottom position
mov  bx,[rownum]  ; Get current row number
cmp  al,bl        ; Compare the two
ja   SKIPPIXEL    ; Jump if floor pixel behind a wall
```

Now we're going to perform some fairly heavy calculations to find the color of the floor pixel visible through the current column in the row. We already know the coordinates of this pixel — for the first pixel, the values are passed to this function in the x and y parameters, and for additional pixels they'll be calculated by adding xinc and yinc to these values — but we need to determine which

texture map, and which position within that texture map, we want to take the color of that pixel from. First, we'll need to use the x,y floor coordinates to index into the floor coordinate array. In C++, indexing into the floor map was a simple matter of dividing the fine x and y coordinates by 64 to get the x,y coordinates of the maze square that the ray is intersecting. We could then retrieve the number of the texture map tile from the appropriate element of the two-dimensional floor array.

It's not *quite* that simple in assembly language. In assembler, we must treat the two-dimensional floor array as a one-dimensional array in which each block of 16 elements represents a row of the maze. Thus even after we've divided the fine coordinates by 64 to get the grid coordinates, we must multiply the y grid coordinate by 16 (the width of the floormap) and add the x grid coordinate to it to get the actual element of the array representing the floor square.

This is complicated by the fact that the x and y fine coordinates, which are currently stored in CPU registers are in 16:16 fixed point format. So we must first use a SHLD instruction to shift the high 16 bits of each of these numbers into a second pair of registers. Then we'll need to shift those numbers back to the right six positions to divide them by 64. This is a chance to save a few CPU cycles, since a 16-position left shift and a 6-position right shift can be combined into a 10-position left shift:

```
shld edi,edx,10          ; (int)x / 64
shld ebx,ecx,10          ; (int)y / 64
```

Next EDI contains the x grid coordinates and EBX contains the y grid coordinate.

The EBX register may have some junk in its high bits, which would not have been removed by this shift. We'll clear them out by using an AND instruction to take the value of the register modulo 16, the breadth of the map:

```
and  ebx,15              ; Clear out junk in EBX
```

Next we'll multiply the y grid coordinate by 16 to get the proper row of the map:

```
shl  ebx,4               ; Multiply y * 16
```

The EDI register may have also have junk in its high bits, so we'll clear that out too:

```
and  edi,15              ; Clear out junk in EDI
```

Finally, we'll add the row offset and the x grid coordinate together to get the offset within the floor array of the square being intersected by the current ray:

```
add  bx,di               ; BX = (int)y/64*16+(int)x/64
```

The EBX register is now pointing at the offset of current floor tile number within the floor array. However, we need to add this value to the offset of the floor array within the current segment before we can actually access the tile. We could do this with an addition operation, but that's redundant. The 80x96 instruction set allows us to use the value in certain registers, including EBC, as an index off of the value of another register. This works much the same way that an array index, such as the x in bigarray[x], works in C++.

The offset of the floor array is in BP and the segment of the array is in ES. We can add the index in EBX to this address and get the number of the current tile like this:

```
mov  al,es:[ebp + ebx]      ; Get tile number in AL
```

The texturelist parameter mentioned earlier is a pointer to an array with 15 entries, each containing the offset into the texture map of the upper-left corner of each of the 15 bitmaps that we're using for the walls, ceilings, and floors. We've stored that pointer in the memory location texture. Now we'll load the pointer into gs:bx:

```
lgs  bx,[texture]           ; Point GS:BX at texture list
```

Using the texture number in AL as an index, we'll get a pointer to the texture map for the current floor square:

```
mov  edi,gs:[ebx + (eax * 4)] ;Get pointer to texture map
```

The EAX * 4 tells the CPU that we're indexing into an array of doubleword (32 bit) values, effecting scaling the index by four.

We'll save the pointer to the texture map temporarily in the memory location *textureptr* for later use:

```
mov  [textureptr],edi       ; Save texture map pointer
```

Meanwhile, we'll calculate the position of pixels within the bitmap that the current ray is striking. In C++, we did this by taking the fine x and y coordinates modulo 64 to get the coordinates within the texture map. Then we multiplied the y coordinate by 320 and added the x coordinate to it to get the actual offset into the texture map file past the offset of the upper-left corner of the bitmap. We can do pretty much the same thing in assembly language. First we convert x and y into integers in the EBX and EDI registers:

```
shld edi,ecx,16
shld ebx,edx,16
```

Then we take x and y modulo 64:

```
and  edi,63
and  ebx,63
```

Next, we multiply y by 320:

```
imul di,320
```

and add x to it:

```
add  di,bx
```

Finally, we load the pointer to the upper-left corner of the bitmap into gs:bx:

```
lgs  bx,[textureptr]
```

and use DI as an index into the texture map array to fetch the color of the current pixel:

```
mov  al,gs:[bx + di]
```

We can lightsource this pixel the same way we did when drawing the walls, by getting a pointer to the lightsourcing array in gs:bx and using AX as an index off of this pointer to get the proper lightsourced color:

```
lgs  bx,[LightIndex]
mov  al,gs:[ebx + eax]
```

The last step is to put the pixel on the display at the address pointed to by fs:si plus the value of the REP parameter:

```
mov  fs:[si + rep],al
```

This brings us to the local SKIPPIXEL label, where we would have ended up if this pixel had been behind a wall:

```
SKIPPIXEL:
```

Before the loop can repeat, we must advance the pointer to the array of wall bottoms to the next column:

```
add  dword ptr [botptr],1
```

and add the x and y increments to the fixed point coordinates of the current floor pixel, to derive the address of the next floor pixel in sequence:

```
add  ecx,[yincrement]
add  edx,[xincrement]
```

And that ends the macro:

```
ENDM
```

To use this macro in the *drawfloorrow()* function, we simply repeat the macro eight times, like this:

```
floor:
```

```
    FLOORLOOP 0
    FLOORLOOP 1
    FLOORLOOP 2
    FLOORLOOP 3
    FLOORLOOP 4
    FLOORLOOP 5
    FLOORLOOP 6
    FLOORLOOP 7
```

Notice that the value of the REP parameter is increased once for each call of the macro, so that the screen index that we used to place the pixel at the proper position on-screen will advance by one for each iteration of the unrolled loop. If we neglected to do this, the floor would appear as a series of diagonal stripes, each separated from the next by seven black background pixels.

We aren't quite done yet. We'll usually want to repeat this loop more than eight times, because our viewport will usually be wider than eight pixels. (Note, however, that it must always be a *multiple* of eight pixels in width, because of the way the loop has been unrolled.) Before the loop can repeat again, we must advance the position of the screen pointer in SI by eight positions:

```
add  si,8
```

During the initialization portion of this function, we created a variable called *colnum* and set it to 0. This variable represents the number of the screen column in which the next pixel is to be drawn. On each iteration, we'll need to add 8 to this value:

```
add  [colnum],8
```

We also need to see if colnum exceeds the width of the screen as specified in the w parameter, which we've stored in memory location width:

```
mov  bx,[colnum]
cmp  bx,[width]
```

If it doesn't, we loop back and execute the eight unrolled floor drawing loops again:

```
jb   floor
```

That's it. We can close up shop and go home:

```
    pop   bp
    ret
_drawfloorrow  ENDP
```

The complete text of the *drawfloorrow()* function is in Listing 12-8. The text of the FLOORLOOP macro is in Listing 12-9.

Listing 12-8 The drawfloorrow() function

```
_drawfloorrow  PROC
     ARG  row:WORD,screenptr:DWORD,texturelist:DWORD,floormap:DWORD,
litelevel:DWORD,bots:DWORD,xinc:DWORD,yinc:DWORD,x:DWORD,y:DWORD,w:WORD
    push  bp                ; Save BP
     mov  bp,sp             ;Set up stack pointer
    ; Move parameters into memory variables:
    mov   bx,w
    mov   [width],bx
    mov   ebx,litelevel
    mov   [lightindex],ebx
    mov   bx,row
    mov   [rownum],bx
    mov   [colnum],0
    mov   ecx,y
    mov   edx,x
    lfs   si,screenptr
    mov   ebx,xinc
    mov   [xincrement],ebx
    mov   ebx,yinc
    mov   [yincrement],ebx
    mov   ebx,texturelist
    mov   [texture],ebx
    mov   ebx,bots
    mov   [botptr],ebx
    les   bp,floormap
    xor   eax,eax          ; Clear the EAX register.

floor:
    FLOORLOOP 0            ; Unroll FLOORLOOP 8 times.
    FLOORLOOP 1
    FLOORLOOP 2
    FLOORLOOP 3
    FLOORLOOP 4
    FLOORLOOP 5
    FLOORLOOP 6
    FLOORLOOP 7
    add  si,8              ; Advance screen pointers
    add  [colnum],8        ; Increase column count
    mov  bx,[colnum]       ; Have we covered entire viewport?
    cmp  bx,[width]
    jb   floor           ; If not, do it again.

    pop   bp                ; Else return to caller.
    ret
_drawfloorrow  ENDP
```

Listing 12-9 The FLOORLOOP macro

```
FLOORLOOP MACRO REP
    LOCAL SKIPPIXEL
    lgs  bx,[botptr]          ; Get pointer to BOTS array
    mov  al,gs:[bx]           ; Get current bottom position
    mov  bx,[rownum]          ; Get current row number
    cmp  al,bl                ; Compare the two
    ja   SKIPPIXEL            ; Jump if floor pixel behind a wall
    shld edi,edx,10           ; (int)x / 64
    shld ebx,ecx,10           ; (int)y / 64
    and  ebx,15               ; Clear out junk in EBX
    shl  ebx,4                ; Multiply y * 16
    and  edi,15               ; Clear out junk in EDI
    add  bx,di                ; BX = (int)y/64*16+(int)x/64
    mov  al,es:[ebp + ebx]     ; Get tile number in AL
    lgs  bx,[texture]          ; Point GS:BX at texture list
    mov  edi,gs:[ebx + (eax * 4)] ;Get pointer to texture map
    mov  [textureptr],edi      ; Save texture-map pointer
    shld edi,ecx,16            ; Calculate (int)y%64*320+x%64
    shld ebx,edx,16
    and  edi,63
    and  ebx,63
    imul di,320
    add  di,bx
    lgs  bx,[textureptr]       ; Get pointer to texture
    mov  al,gs:[bx + di]       ; Get pixel color
    lgs  bx,[LightIndex]       ; Point to lightsource table
    mov  al,gs:[ebx + eax]     ; Get lightsourced color
    mov  fs:[si + rep],al      ; Put in on screen
SKIPPIXEL:
    add  dword ptr [botptr],1  ; Advance bottom pointer
    add  ecx,[yincrement]      ; Add increments to get
    add  edx,[xincrement]      ; next pixel coordinates
ENDM
```

The *drawceilrow* function is so nearly identical to the *drawfloorrow* function that there's no need to print it here, though it can be found in the RAY.ASM file in the OPTIMIZE directory on the accompanying disk. The only significant difference is that an array of the top positions of walls is used instead of an array of bottom positions to determine if a pixel needs to be drawn.

The Optimized draw_maze() Function

The optimized version of the *draw_maze()* function is essentially the same as the lightsourced version in chapter 11. The main difference is that fixed point numbers and operations have been substituted for all *float* values (with one exception, involving the use of the arctangent function, which is difficult to make table-driven). In addition, calls to the three assembly language functions have been substituted in place of the code that drew the wall columns and the floor and ceiling pixels.

I won't detail every change, but I'll give you some examples. For instance, the original lightsourced *draw_maze()* function initialized the floating point variables x and y to the position of the viewer like this:

```
float x=xview;
float y=yview;
```

In the fixed point version, the float variables x and y become the fixed point variables fix_x and fix_y. (Note the use of the prefix *fix* as a simple method of distinguishing fixed point variables from non-fixed point variables.) In order to initialize these fixed point variables, the values of xview and yview must first be converted to fixed point:

```
long fix_x=(long)xview<<SHIFT;
long fix_y=(long)yview<<SHIFT;
```

Similarly, when the original lightsourced version rotated a ray through a screen pixel, it did it like this:

```
int x2 = -1024 * (sin(radians));
int y2 = 1024 * (cos(radians));
```

When the fixed point version does this, it uses a table look-up and our fixed point multiplication function:

```
int x2 = fixmul(-256L<<SHIFT,SIN(degrees))>>SHIFT;
int y2 = fixmul(256L<SHIFT,COS(degrees))>SHIFT;
```

The shifts in this equation may seem a bit confusing. The first converts 256 (an arbitrary length for the line to be rotated, as was the 1024 in the earlier version) into a fixed point number. This could be done by hand, but it's easier to let the computer do it, since most compilers will perform this operation at compile time rather than runtime, so that we won't waste extra CPU cycles. The shift at the end of the line converts the value of the entire expression back to an integer, for storage in x2 and xy.

Other changes in the function follow this basic design. The complete text of the optimized *draw_maze()* function appears in Listing 12-10.

Listing 12-10 The optimized draw_maze() function

```
int getdistance(long degrees,int column_angle,long x,long y,
                int xview,int yview);

unsigned char bottoms[320],tops[320];

void draw_maze(map_type map,map_type floor,map_type ceiling,
               char far *screen,int xview,int yview,
               long viewing_angle,int viewer_height,
               int ambient_level,unsigned char far * textmaps,
               unsigned char litesource[MAXLIGHT+1][PALETTESIZE],
               unsigned char *litelevel)

// Draws a raycast image in the viewport of the maze represented
// in array MAP[], as seen from position XVIEW, YVIEW by a
// viewer looking at angle VIEWING_ANGLE where angle 0 is due
// north. (Angles are measured in radians.)

{
  // Variable declarations:
  int sy,offset;                    // Pixel y position and offset
  long fix_xd,fix_yd;               // Distance to next wall in x and y
  int grid_x,grid_y;                // Coordinates of x and y grid lines
  long fix_xcross_x,fix_xcross_y;   // Ray intersection coordinate
  long fix_ycross_x,fix_ycross_y;
  unsigned int xdist,ydist;         // Distance to x and y grid lines
  int xmaze,ymaze;                  // Map location of ray collision
  int distance;                     // Distance to wall along ray
  int tmcolumn;                     // Column in texture map
  int top,bot;
  long fix_yratio;

  // Loop through all columns of pixels in viewport:
  for (int column=VIEWPORT_LEFT; column<=VIEWPORT_RIGHT;
          column++) {

    // Calculate horizontal angle of ray relative to
    //  center ray:
    int column_angle=atan((float)(column-160)
                / VIEWER_DISTANCE)*(NUMBER_OF_DEGREES/6.28);
    if (column_angle<0) column_angle+=NUMBER_OF_DEGREES;
    if (column_angle>NUMBER_OF_DEGREES-1) column_angle-=
                  NUMBER_OF_DEGREES;

    // Calculate angle of ray relative to maze coordinates
    int degrees=viewing_angle+column_angle;
    if (degrees>NUMBER_OF_DEGREES-1) degrees-=NUMBER_OF_DEGREES;
```

continued on next page

continued from previous page

```
    // Rotate endpoint of ray to viewing angle:
    int x2 = fixmul(-256L<<SHIFT,SIN(degrees))>>SHIFT;
    int y2 = fixmul(256L<SHIFT,COS(degrees))>SHIFT;

    // Translate relative to viewer's position:
    x2+=xview;
    y2+=yview;

    // Initialize ray at viewer's position:
    long fix_x=(long)xview<<SHIFT;
    long fix_y=(long)yview<<SHIFT;

    // Find difference in x,y coordinates along ray:
    long xdiff=x2-xview;
    long ydiff=y2-yview;

    // Cheat to avoid divide-by-zero error:
    if (xdiff==0) xdiff=1;
    if (ydiff==0) ydiff=1;

    // Get slope of ray:
    long fix_slope = fixdiv((long)ydiff<<SHIFT,
                            (long)xdiff<<SHIFT);

    // Cast ray from grid line to grid line:
    for (;;) {

      // If ray direction positive in x, get next x grid line:
      if (xdiff>0) grid_x=((fix_x>>SHIFT) & 0xffc0) + 64;

      // If ray direction negative in x, get last x grid line:
      else grid_x=((fix_x>>SHIFT) & 0xffc0) - 1;

      // If ray direction positive in y, get next y grid line:
      if (ydiff>0) grid_y=((fix_y>>SHIFT) & 0xffc0) + 64;

      // If ray direction negative in y, get last y grid line:
      else grid_y=((fix_y>>SHIFT) & 0xffc0) - 1;

      // Get x,y coordinates where ray crosses x grid line:
      fix_xcross_x = (long) grid_x << SHIFT;
      fix_xcross_y = fix_y + fixmul(fix_slope,((long)grid_x
                          <<SHIFT)-fix_x);

      // Get x,y coordinates where ray crosses y grid line:
      fix_ycross_x=fix_x+fixdiv((((long)grid_y<<SHIFT)-fix_y),
              fix_slope);
      fix_ycross_y=(long)grid_y<<SHIFT;

      // Get distance to x grid line:
      fix_xd=fix_xcross_x-fix_x;
      fix_yd=fix_xcross_y-fix_y;
```

```
    distance=getdistance(degrees,column_angle,
       fix_xcross_x,fix_xcross_y,xview,yview);
    xdist=distance;

    // Get distance to y grid line:
    fix_xd=fix_ycross_x-fix_x;
    fix_yd=fix_ycross_y-fix_y;
    long temp_f1=fixmul(fix_xd,fix_xd);
    long temp_f2=fixmul(fix_yd,fix_yd);
    distance=getdistance(degrees,column_angle,
       fix_ycross_x,fix_ycross_y,xview,yview);
    ydist=distance;

    // If x grid line is closer...
    if (xdist<ydist) {

      // Calculate maze grid coordinates of square:
      xmaze=fix_xcross_x>>22;
      ymaze=fix_xcross_y>>22;

      // Set x and y to point of ray intersection:
      fix_x=fix_xcross_x;
      fix_y=fix_xcross_y;

      // Find relevant column of texture map:
      tmcolumn = (fix_y>>SHIFT) & 0x3f;

      // Is there a maze cube here? If so, stop looping:
      if (map[xmaze][ymaze]) break;
    }
    else { // If y grid line is closer:

      // Calculate maze grid coordinates of square:
      xmaze=fix_ycross_x>>22;
      ymaze=fix_ycross_y>>22;

      // Set x and y to point of ray intersection:
      fix_x=fix_ycross_x;
      fix_y=fix_ycross_y;

      // Find relevant column of texture map:
      tmcolumn = (fix_x>>SHIFT) & 0x3f;

      // Is there a maze cube here? If so, stop looping:
      if (map[xmaze][ymaze]) break;
    }
  }

  // Get distance from viewer to intersection point:
  distance=getdistance(degrees,column_angle,
     fix_x,fix_y,xview,yview);

  if (distance==0) distance=1;
```

continued on next page

continued from previous page

```
// Calculate visible height of wall:
int height = VIEWER_DISTANCE * WALL_HEIGHT / distance;
if (!height) height=1;

// Calculate bottom of wall on screen:
bot = VIEWER_DISTANCE * viewer_height
          / distance + VERT_CENTER;

// Calculate top of wall on screen:
top = bot - height + 1;

// Initialize temporary offset into texture map:
      int t=tmcolumn*320+IMAGE_HEIGHT;

// If top of current vertical line is outside of
// viewport, clip it:
long dheight=height;
long iheight=IMAGE_HEIGHT;
long fix_yratio=fixdiv((long)WALL_HEIGHT<<SHIFT,
      (long)height<<SHIFT);
 if (top < VIEWPORT_TOP) {
  int diff=VIEWPORT_TOP-top;
   dheight-=diff;
  iheight -= (diff*fix_yratio)>>SHIFT;
   top=VIEWPORT_TOP;
 }
 if (bot >= VIEWPORT_BOT) {
  int diff=bot-VIEWPORT_BOT;
  dheight -= diff;
  iheight -= diff*fix_yratio>>SHIFT;
  t-=(diff*fix_yratio)>>SHIFT;
  bot=VIEWPORT_BOT;
}

// Save top and bottom in arrays:
tops[column]=top;
bottoms[column]=bot;

// Point to video memory offset for top of line:
 offset = bot * 320 + column;

// Which graphics tile are we using?
int tile=map[xmaze][ymaze]-1;

// Find offset of tile and column in bitmap:
unsigned int tileptr=(tile/5)*320*IMAGE_HEIGHT+(tile%5)
  *IMAGE_WIDTH+t;


long fix_increment=fixdiv(iheight<<SHIFT,dheight<<SHIFT);
int level=litelevel[distance]+ambient_level;
if (level>MAXLIGHT) level=MAXLIGHT;
```

```
  drawwall(&(screen[offset]),&(textmaps[tileptr]),dheight,
              fix_increment,&(litesource[level][0]));
}

  // Step through floor pixels:
  for (int row=VERT_CENTER+5; row<=VIEWPORT_BOT; row++) {

    // Calculate horizontal angle of leftmost column relative
    // to center ray:
    int column_angle=atan((float)(VIEWPORT_LEFT-HORIZ_CENTER)
                / VIEWER_DISTANCE)*(NUMBER_OF_DEGREES/6.28);
    if (column_angle<0) column_angle+=NUMBER_OF_DEGREES;
    if (column_angle>NUMBER_OF_DEGREES-1) column_angle-=
                    NUMBER_OF_DEGREES;

    // Calculate angle of ray relative to maze coordinates
    int degrees=viewing_angle+column_angle;
    if (degrees>NUMBER_OF_DEGREES-1) degrees-=
                  NUMBER_OF_DEGREES;

    // Get ratio of viewer's height to pixel height:
    long fix_ratio=fixdiv((long)viewer_height<<SHIFT,
                          (long)(row-VERT_CENTER)<<SHIFT);

    // Get distance to visible pixel:
    long fix_distance=fixdiv(fix_ratio*VIEWER_DISTANCE,
                       COS(column_angle));

    // Rotate distance to ray angle:
    int left_x = - (fixmul(fix_distance,SIN(degrees))>>SHIFT);
    int left_y = fixmul(fix_distance,COS(degrees))>>SHIFT;

    // Translate relative to viewer coordinates:
    left_x+=xview;
    left_y+=yview;

    // Calculate horizontal angle of rightmost column relative
    // to center ray:
    column_angle=atan((float)(VIEWPORT_RIGHT-HORIZ_CENTER)
                / VIEWER_DISTANCE)*(NUMBER_OF_DEGREES/6.28);
    if (column_angle<0) column_angle+=NUMBER_OF_DEGREES;
    if (column_angle>NUMBER_OF_DEGREES-1) column_angle-=
           NUMBER_OF_DEGREES;

    // Calculate angle of ray relative to maze coordinates
    degrees=viewing_angle+column_angle;
    if (degrees>NUMBER_OF_DEGREES-1) degrees-=
            NUMBER_OF_DEGREES;

    // Get ratio of viewer's height to pixel height:
    fix_ratio=fixdiv((long)viewer_height<<SHIFT,
                    (long)(row-VERT_CENTER)<<SHIFT);
```

continued on next page

continued from previous page

```
    // Get distance to visible pixel:
    fix_distance=fixdiv(fix_ratio*VIEWER_DISTANCE,
           COS(column_angle));

    // Rotate distance to ray angle:
    int right_x = - (fixmul(fix_distance,SIN(degrees))>>SHIFT);
    int right_y = fixmul(fix_distance,COS(degrees))>>SHIFT;

    // Translate relative to viewer coordinates:
    right_x+=xview;
    right_y+=yview;

    // Calculate stepping increment:
    long fix_x_increment=fixdiv((long)(right_x-left_x)<<SHIFT,
                 (long)(VIEWPORT_RIGHT-VIEWPORT_LEFT)<<SHIFT);
    long fix_y_increment=fixdiv((long)(right_y-left_y)<<SHIFT,
                 (long)(VIEWPORT_RIGHT-VIEWPORT_LEFT)<<SHIFT);
    long fix_x=(long)left_x<<SHIFT;
    long fix_y=(long)left_y<<SHIFT;

    int level=litelevel[fix_distance>>SHIFT]+ambient_level;
    if (level>MAXLIGHT) level=MAXLIGHT;

    drawfloorrow(row,&(screen[row*320+VIEWPORT_LEFT]),
                 &(texturelist[0]),&(floor[0][0]),
                 &(litesource[level][0]),
                 &(bottoms[VIEWPORT_LEFT]),fix_x_increment,
                 fix_y_increment,fix_x,fix_y,VIEWPORT_WIDTH);
}

// Step through ceiling pixels:
for (row=VERT_CENTER-5; row>=VIEWPORT_TOP; ñrow) {

    // Calculate horizontal angle of leftmost column relative
    // to center ray:
    int column_angle=atan((float)(VIEWPORT_LEFT-HORIZ_CENTER)
                / VIEWER_DISTANCE)*(NUMBER_OF_DEGREES/6.28);
    if (column_angle<0) column_angle+=NUMBER_OF_DEGREES;
    if (column_angle>NUMBER_OF_DEGREES-1) column_angle-=
           NUMBER_OF_DEGREES;

    // Calculate angle of ray relative to maze coordinates
    int degrees=viewing_angle+column_angle;
    if (degrees>NUMBER_OF_DEGREES-1) degrees-=
          NUMBER_OF_DEGREES;

    // Get ratio of viewer's height to pixel height:
    long fix_ratio=fixdiv((long)(WALL_HEIGHT-viewer_height)
                 <<SHIFT,(long)(VERT_CENTER-row)<<SHIFT);

    // Get distance to visible pixel:
```

```
long fix_distance=fixdiv(fix_ratio*VIEWER_DISTANCE,
                  COS(column_angle));

// Rotate distance to ray angle:
int left_x = - (fixmul(fix_distance,SIN(degrees))>>SHIFT);
int left_y = fixmul(fix_distance,COS(degrees))>>SHIFT;

// Translate relative to viewer coordinates:
left_x+=xview;
left_y+=yview;

// Calculate horizontal angle of rightmost column relative
// to center ray:
column_angle=atan((float)(VIEWPORT_RIGHT-HORIZ_CENTER)
            / VIEWER_DISTANCE)*(NUMBER_OF_DEGREES/6.28);
if (column_angle<0) column_angle+=NUMBER_OF_DEGREES;
if (column_angle>NUMBER_OF_DEGREES-1) column_angle-=
           NUMBER_OF_DEGREES;

// Calculate angle of ray relative to maze coordinates
degrees=viewing_angle+column_angle;
if (degrees>NUMBER_OF_DEGREES-1) degrees-=
       NUMBER_OF_DEGREES;

// Get ratio of viewer's height to pixel height:
fix_ratio=fixdiv((long)(WALL_HEIGHT-viewer_height)<<SHIFT,
     (long)(VERT_CENTER-row)<<SHIFT);

// Get distance to visible pixel:
fix_distance=fixdiv(fix_ratio*VIEWER_DISTANCE,
      COS(column_angle));

// Rotate distance to ray angle:
int right_x = -(fixmul(fix_distance,SIN(degrees))>>SHIFT);
int right_y = fixmul(fix_distance,COS(degrees))>>SHIFT;

// Translate relative to viewer coordinates:
right_x+=xview;
right_y+=yview;

// Calculate stepping increment:
long fix_x_increment=fixdiv((long)(right_x-left_x)<<SHIFT,
            (long)(VIEWPORT_RIGHT-VIEWPORT_LEFT)<<SHIFT);
long fix_y_increment=fixdiv((long)(right_y-left_y)<<SHIFT,
            (long)(VIEWPORT_RIGHT-VIEWPORT_LEFT)<<SHIFT);
long fix_x=(long)left_x<<SHIFT;
long fix_y=(long)left_y<<SHIFT;

int level=litelevel[fix_distance>>SHIFT]+ambient_level;
if (level>MAXLIGHT) level=MAXLIGHT;

drawceilrow(row,&(screen[row*320+VIEWPORT_LEFT]),
           &(texturelist[0]),&(ceiling[0][0]),
```

continued on next page

continued from previous page

```
                    &(litesource[level][0]),
                    &(tops[VIEWPORT_LEFT]),fix_x_increment,
                    fix_y_increment,fix_x,fix_y,VIEWPORT_WIDTH);
    }
}

int getdistance(long degrees,int column_angle,long x,long y,
                    int xview,int yview)
{
  long trig,fix_ny,fix_nx;
  int distance;

  int c=degrees/512;
  if (((c&3)==0)||((c&3)==3))
    distance=abs(fixdiv((y-((long)yview<SHIFT))>SHIFT,
     COS(degrees-1024*c*2))*COS(column_angle)>>SHIFT);
  else
    distance=abs(fixdiv((x-((long)xview<SHIFT))>SHIFT,
     SIN(degrees-1024*c*2))*COS(column_angle)>>SHIFT);
  return(distance);
}
```

The Optdemo Program

We need a demonstration program to show off the capabilities of our new engine. This demo will be based on the demo program used in our earlier animations. It will use the event manager that we developed back in chapter 3 to detect input events, which will allow the user to explore the ray-cast maze.

We really need only one new capability here. In our earlier animations, we used the built-in BIOS timer to set the frame rate of our programs. But the BIOS timer doesn't have a high enough resolution for the timing that we need now; it doesn't slice time into small enough fractions of a second.

Instead, we'll use the HTIMER routines developed by Mark Betz for my earlier book, *Flights of Fantasy* (Waite Group Press, 1993). These timer functions, designed to be called from C++, allow us to time events down to millionths of a second. The HTIMER routines can be found on the disk in the file HTIMER.CPP, along with the file HTIMER.DOC.

The timing will work a bit differently here too. Instead of maintaining a steady number of frames per second, we'll try to eke out as many frames a second as we can. But on faster machines, less time will pass between the frames — and that will be reflected in our demo in that we'll move the viewer a small distance between frames the faster the frame rate gets. We'll base this movement distance on the time between frames returned by the HTIMER routines.

As a result, the speed with which users move through the maze will be independent of the speed of their machine. Owners of slow machines will cover exactly as much ground per second as will owners of fast machines. However, those with fast machines will see more frames per second, making the animation appear much smoother and more realistic.

In order to use the HTIMER routines, we must create an instance of an HTIMER object:

```
HTimer timer1;
```

To start the timer running, we invoke the timerOn() function, which requires no parameters:

```
timer1.timerOn();
```

When we want to know how much time has passed since we turned the timer on, we use the timerOff() function, passing its value to a long variable:

```
elapsed_time = timer1.timerOff();
```

The time is returned in microseconds — that is, millionths of a second.

This gives us a simple way to time a frame: Use timerOn() before the frame is drawn, timerOff() before we start the next frame. We can then use the value of elapsed_time to calculate how far the viewer should move between frames, by creating a pair of constants called DISTANCE_PER_SECOND and ROTATION_PER_SECOND to represent the number of coordinates that we want the user to traverse in one second and the number of degrees (in our 4096-degree system) that we want the user to rotate in a second, respectively:

```
const DISTANCE_PER_SECOND=128;
const ROTATION_PER_SECOND=1024;
```

We can then calculate the actual distance traveled by dividing the distance per second by one million to get the distance per microsecond and multiplying it by the number of microseconds in elapsed_time.

```
long new_y=(long)(DISTANCE_PER_SECOND/1000000.0
     *elapsed_time)<<SHIFT;
```

This value is converted into a fixed point number to facilitate the next step in determining the viewer's new position, which is projecting an imaginary point in front of the viewer, rotating to the viewer's orientation, and scaling it to the viewer's position, just as we rotate and scale rays in the ray-casting function:

```
xview+=(fixmul(new_x,COS(viewing_angle))
     -fixmul(new_y,SIN(viewing_angle)))>>16;
yview+=(fixmul(new_y,COS(viewing_angle))
     +fixmul(new_x,SIN(viewing_angle)))>>16;
```

To move the viewer backwards, we just make the new_y value negative.

A similar series of operations can be used to rotate the viewer. We divide ROTATION_PER_SECOND by 1 million, then multiply it by elapsed_time. The result is added to viewing_angle, the variable that tells us which way the viewer is facing:

```
viewing_angle+=((float)ROTATION_PER_SECOND/1000000.0
     *elapsed_time);
```

In case the angle has become less than 0 or greater than 4095, we check for wraparound:

```
if (viewing_angle>NUMBER_OF_DEGREES-1)
     viewing_angle-=NUMBER_OF_DEGREES;
```

The rest of the optimized maze demo is pretty much the same as the earlier demos, except that we've added a pair of functions called *AddFrameTime()* and *PrintFrameRate(),* adapted from similar functions written by Mark Betz for *Flights of Fantasy.* The first of these functions maintains an array of 500 frame times, as calculated between frames by the HTIMER routines. When the program terminates, *PrintFrameRate()* is called to average these values and calculate the average time between frames and the average frame rate per second, and prints these values on the display.

The complete text of the OPTDEMO.CPP module appears in Listing 12-11.

Listing 12-11 The OPTDEMO.CPP module

```
// OPTDEMO.CPP
//
// Calls optimized ray-casting function to draw lightsourced
// view of maze.
//
// Written by Christopher Lampton for
// Gardens of Imagination (Waite Group Press, 1994)
#include  <stdio.h>
#include  <dos.h>
#include  <conio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "screen.h"
#include  "optdemo.h"
#include  "pcx.h"
#include  "raycast.h"
#include  "evntmgr1.h"
#include  "bitmap.h"
#include  "io.h"
```

```
#include  "trig.h"
#include  "types.h"
#include  "htimer.h"
#include  "fixpoint.h"

// Function prototypes:
void PrintFrameRate();
void near AddFrameTime(long elapsedtime);

// Constant definitions:
const MAXFRAMES=500;
const NUM_IMAGES=15;
const float MULTIPLIER=3;
const DISTANCE_PER_SECOND=128;
const ROTATION_PER_SECOND=1024;
const WHICH_EVENTS=KEYBOARD_EVENTS+MOUSE_EVENTS;

map_type walls={
 { 2, 2, 2, 2, 7, 4, 7, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 9, 2, 0, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2},
 { 9, 0, 0, 0, 0, 0, 0, 2, 8, 0, 0, 0, 0, 0, 0, 2},
 { 9, 0, 0, 0, 0, 0, 0, 0, 8, 6, 0, 0, 0, 0, 0, 2},
 { 9, 0, 0, 0, 0, 0, 0, 0, 0, 6, 2, 2, 0, 0, 0, 2},
 { 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 2},
 { 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 9, 9, 0, 0, 2},
 { 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 2},
 { 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 9, 9, 0, 0, 2},
 { 5, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 0, 0, 0, 2},
 { 2, 2, 2, 0, 0, 0, 2, 2, 0, 2, 0, 0, 0, 0, 0, 2},
 { 7, 0, 0, 0, 0, 0, 2, 2, 0, 2, 0, 0, 0, 0, 0, 2},
 { 7, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2},
 { 7, 0, 0, 0, 0, 0, 0, 2, 0, 2, 0, 0, 0, 0, 0, 2},
 { 7, 7, 7, 7, 7, 0, 0, 2, 0, 2, 0, 0, 0, 0, 0, 2},
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
};

map_type flor={
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 2, 5, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 5, 5, 5, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 5, 5, 5, 5, 5, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 5, 5, 5, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 2, 5, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
```

continued on next page

continued from previous page

```
 { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
};

map_type ceiling={
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9},
 { 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9}
};

pcx_struct textmaps,bg;  // PCX files for tiles and background
FILE *handle;            // Handle for lightsourcing file
unsigned char litetable[MAXLIGHT+1][PALETTESIZE];
unsigned char *litelevel;
unsigned char *texturelist[15];

long viewing_angle=13;
long intensity=MAXLIGHT;
long viewer_height=32;
long ambient_level=MAXLIGHT;
long xview=8*64+32;
long yview=8*64+32;

HTimer timer1;                  // instance of HTimer object
HTimer* timer2;                 // pointer to HTimer object
long elapsed_time;
unsigned long frame[MAXFRAMES];
int frameptr=0;
int framewrap=0;

void main(int argc,char* argv[])
{
  event_struct events;
  unsigned far char *screen_buffer;

  // Read arguments from command line if present:
  if (argc>=2) intensity=atof(argv[1]);
  if (argc>=3) ambient_level=atof(argv[2]);

  // Load texture map images:
```

```
if (loadPCX("walls.pcx",&textmaps)) exit(1);

// Load background screen image:
if (loadPCX("demobg8.pcx",&bg)) exit(1);

// Create texture list array:
for (unsigned int i=0; i<NUM_IMAGES; i++) {
  texturelist[i]=&(textmaps.image[(i/5)*320*64+i%5*64]);
}

// Load lightsourcing tables:
if ((handle=fopen("litesorc.dat","rb"))==NULL) {
  perror("Error");
  exit;
}
fread(litetable,MAXLIGHT+1,PALETTESIZE,handle);
fclose(handle);

// Initialize array of light levels:
litelevel=new unsigned char[MAXDISTANCE];

// Calculate light intensities for all possible
// distances:
for (int distance=1; distance<MAXDISTANCE; distance++) {
  float ratio=(float)intensity/distance*MULTIPLIER;
  if (ratio>1.0) ratio=1.0;
  litelevel[distance]=ratio*MAXLIGHT;
}

// Point variable at video memory:
char far *screen=(char far *)MK_FP(0xa000,0);

   // Create offscreen video buffer:
   if ((screen_buffer=new unsigned char [64000])==NULL) {;
  perror("Error");
  exit;
}

// Allocate memory for timer object:
if ((timer2 = new HTimer()) == NULL) {
  perror("Initialization of timer object failed");
  exit;
}

   // Initialize event manager:
   init_events();

   // Calibrate the user's joystick:
   if (WHICH_EVENTS & JOYSTICK_EVENTS) {
        printf("\nCenter your joystick and press button ");
        printf("one.\n");
        setcenter(); // Calibrate the center position
```

continued on next page

continued from previous page

```
            printf("Move your joystick to the upper lefthand ");
            printf("corner and press button one.\n");
            setmin();    // Calibrate the minimum position
            printf("Move your joystick to the lower righthand ");
            printf("corner and press button one.\n");
            setmax();    // Calibrate the maximum position
        }

    // Save previous video mode:
    int oldmode=*(int *)MK_FP(0x40,0x49);

    // Set mode 13h:
    setmode(0x13);

    // Set the palette:
    setpalette(textmaps.palette);

    // Clear the screen:
    cls(screen_buffer);

    // Copy background into screen buffer:
    for (i=0; i<64000; i++)
      screen_buffer[i]=bg.image[i];

    // Draw initial view of maze:
    draw_maze(walls,flor,ceiling,screen_buffer,xview,yview,viewing_angle,
              viewer_height,ambient_level,textmaps.image,
              litetable,litelevel);

    // Put screen buffer contents on screen:
    putwindow(0,0,320,200,screen_buffer);

    events.quit_game=0;

    timer1.timerOn();

       // Let's go for a walk in the maze:
       while(!events.quit_game) {

       // Draw the maze in screen buffer:
      clrwin(VIEWPORT_LEFT,VIEWPORT_TOP,VIEWPORT_WIDTH,
             VIEWPORT_HEIGHT,screen_buffer);

      elapsed_time = timer1.timerOff();
      AddFrameTime(elapsed_time);
      timer1.timerOn();

      // Put first frame into buffer:
draw_maze(walls,flor,ceiling,screen_buffer,xview,yview,viewing_angle,
              viewer_height,ambient_level,textmaps.image,
              litetable,litelevel);
```

```
// Move screen buffer to screen:
putwindow(VIEWPORT_LEFT,VIEWPORT_TOP,VIEWPORT_WIDTH,
         VIEWPORT_HEIGHT,screen_buffer);

 // Move viewer according to input events:
 getevent(WHICH_EVENTS,&events);

// Did a go forward event occur?
if (events.go_forward) {

  // If so, assume viewer is standing at origin of map...
  long new_x=0;

  // ...facing north. Calculate destination by checking
  // timer to see what fraction of a second has passed
  // since last frame of animation:
  long new_y=(long)(DISTANCE_PER_SECOND/1000000.0
              *elapsed_time)<<16;

  // Rotate destination to actual viewer heading and
  // translate relative to actual viewer position:
  xview+=(fixmul(new_x,COS(viewing_angle))
          -fixmul(new_y,SIN(viewing_angle)))>>16;
  yview+=(fixmul(new_y,COS(viewing_angle))
          +fixmul(new_x,SIN(viewing_angle)))>>16;
}

// Did a go back event occur?
if (events.go_back) {

  // Repeat procedure for forward event...
  long new_x=0;

  // ...but make the distance negative, to the rear
  // of the viewer:
  long new_y=-(long)(DISTANCE_PER_SECOND/1000000.0
                      *elapsed_time)<<16;

  // Rotate and translate destination as above:
  xview+=(fixmul(new_x,COS(viewing_angle))
          -fixmul(new_y,SIN(viewing_angle)))>>16;
  yview+=(fixmul(new_y,COS(viewing_angle))
          +fixmul(new_x,SIN(viewing_angle)))>>16;
}

// Did a go right event occur?
if (events.go_right) {

  // If so, calculate new angle based on elapsed time since
  // last frame of animation:
```

continued on next page

continued from previous page

```
      viewing_angle+=((float)ROTATION_PER_SECOND/1000000.0
                        *elapsed_time);

      // Wrap around to 0 if angle exceeds maximum degrees:
      if (viewing_angle>NUMBER_OF_DEGREES-1)
           viewing_angle-=NUMBER_OF_DEGREES;
    }

    // Did a go left event occur?
    if (events.go_left) {

      // If so, calculate new angle based on elapsed time since
      // last frame of animation:
      viewing_angle-=((float)ROTATION_PER_SECOND/1000000.0
                        *elapsed_time);

      // Wrap around to maximum degree if angle less than zero:
      if (viewing_angle<0) viewing_angle+=NUMBER_OF_DEGREES;
    }
  }

  // Reset video mode and exit:
  setmode(oldmode);
  PrintFrameRate();
}

void PrintFrameRate()
{
 int numframes;
 unsigned long looptime = 0;

 if (framewrap) numframes = MAXFRAMES;
 else numframes = frameptr;
 cprintf("Average time per frame (ms):");
 if (numframes) {
   for (int i = 0; i < numframes; i++) looptime += frame[i];
   looptime /= numframes;
   cprintf(" %i\r\n", looptime);
   printf("Average frames per second: %d",1000/looptime);
 }
 else cprintf(" timer disabled\r\n");
}

void near AddFrameTime(long elapsedtime)
{
  frame[frameptr] = (elapsedtime/1000);
  frameptr++;
  if (frameptr == MAXFRAMES) {
    frameptr = 0;
    framewrap = -1;
  }
}
```

Running OPTDEMO.EXE

To run OPTDEMO, go to the OPTIMIZE directory and type:

```
OPTDEMO
```

You'll see a view of the maze similar to ones displayed by our earlier ray-casting programs, but this time you can move through the maze using the cursor keys. The ↑ cursor key moves forward, the ↓ cursor key moves backwards, the ← cursor key rotates left, and the → cursor key rotates right. The ESC key terminates the animation.

OPTDEMO accepts two parameters: the first a value in the range 0 to 32 representing the level of the viewer's torch, the second a value in the same range representing the level of the ambient lighting. Both of these are set to 32 — full brightness — by default, but you can play with other light levels by setting these to lower values.

Figures 12-4a–d show views of the maze from various angles and lighting. Have fun exploring!

Optimization Review

We've covered a lot of material in this chapter. Let's look back briefly over the optimization techniques that were discussed:

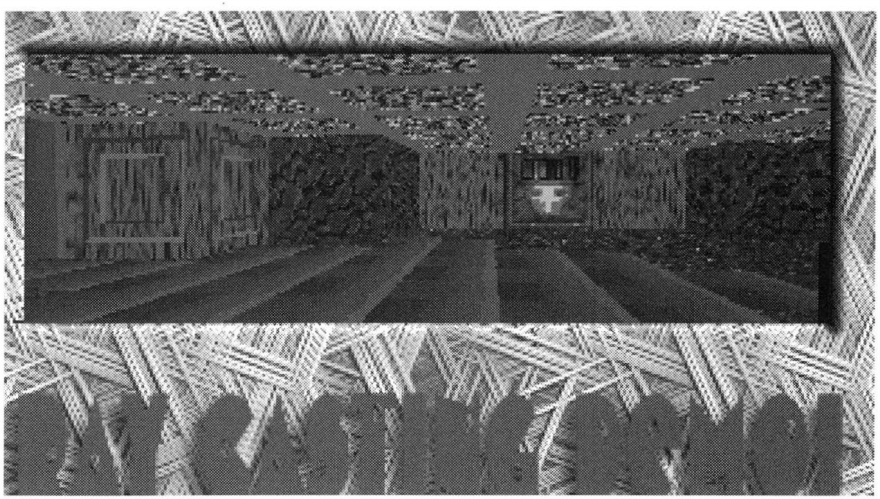


Figure 12-4a View of the maze from OPTDEMO

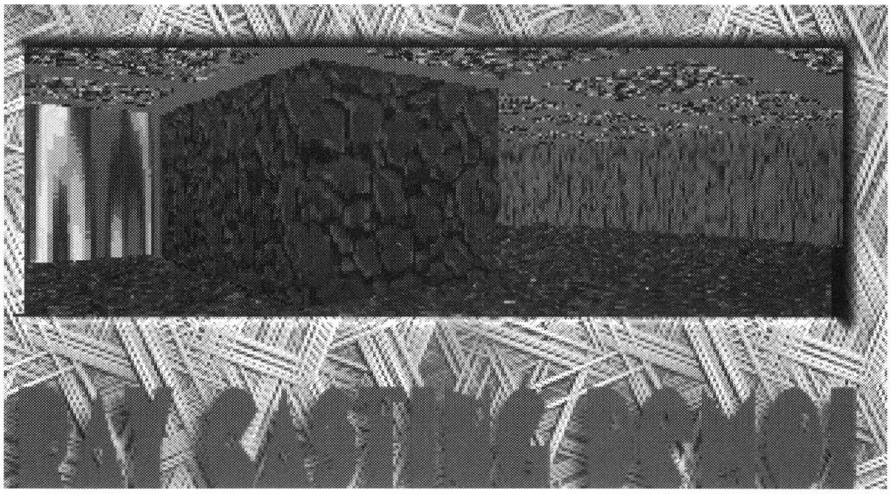


Figure 12-4b Second view of the maze from OPTDEMO

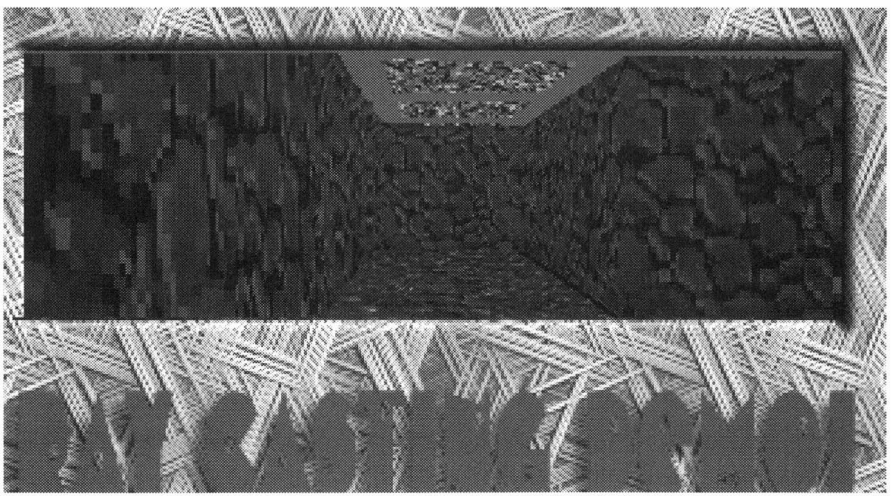


Figure 12-4c Third view of the maze from OPTDEMO

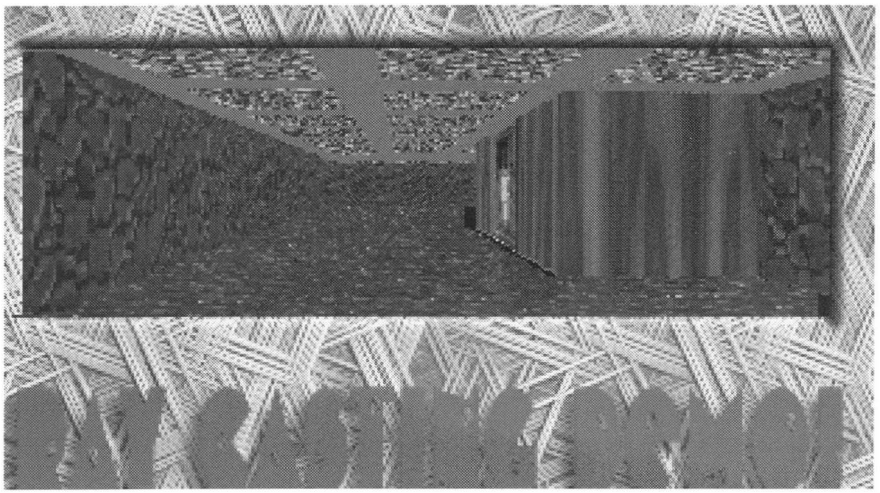


Figure 12-4d Fourth view of the maze from OPTDEMO

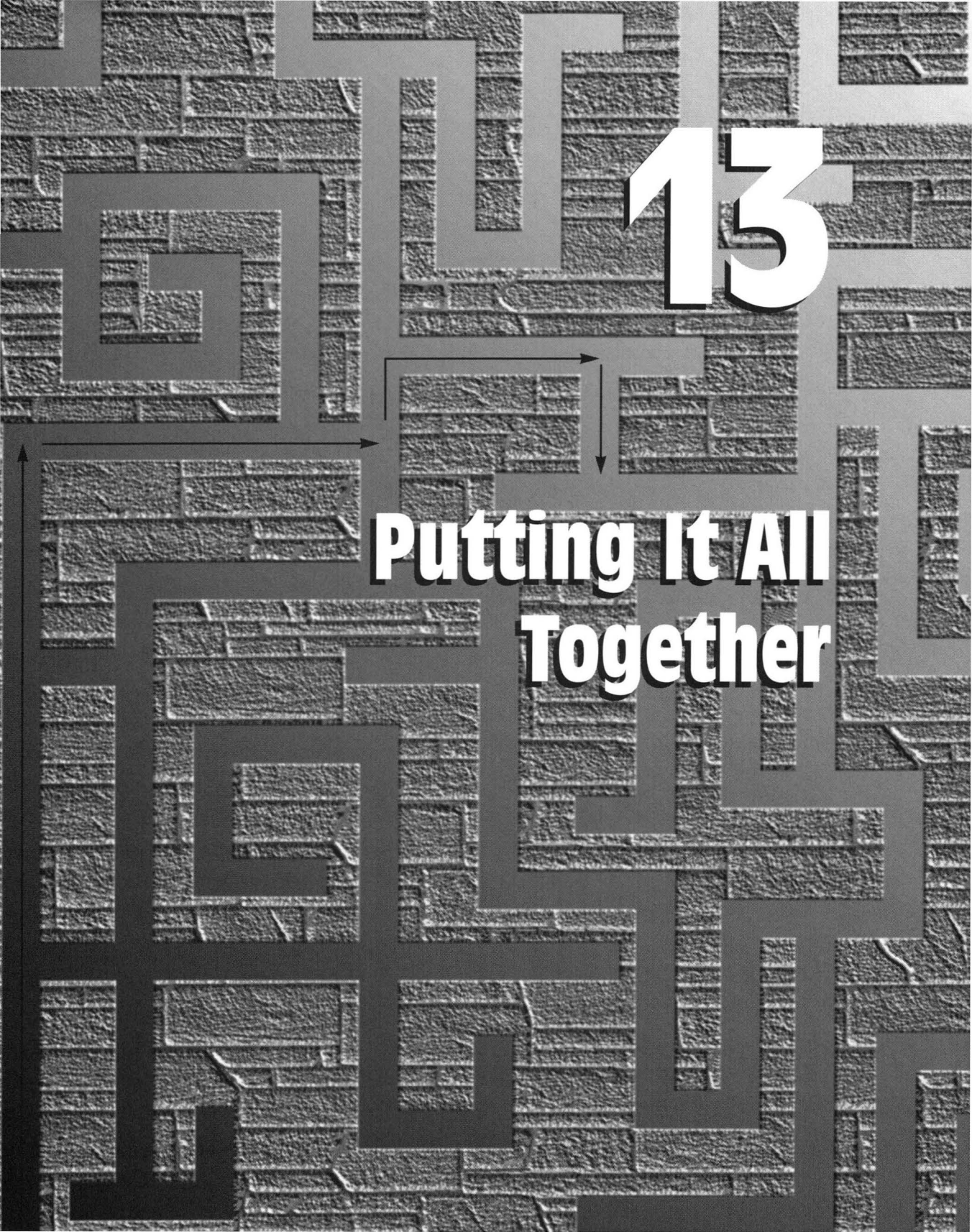
- *Translating into Assembly Language*. Although this isn't a trick that beginning programmers should attempt on their first foray into game programming, medium-level and advanced programmers will find that this is a reliable way of achieving those last few ounces of animation speed. Remember, however, that this technique should be restricted only to the most time critical portions of your program.

- *Unrolling the Loop*. By "unrolling" a loop — that is, converting a loop to a sequence of repeated instructions — extremely time-critical inner loops can be speeded up significantly, especially if the number of instructions in the loop is relatively small. However, it isn't necessary to unroll a loop fully. Unrolling it eight times and placing the resulting code inside another loop will give you most of the benefits of an unrolled loop without many of the associated problems, such as bloated code and overflowing CPU caches. Remember to use macros to reduce both typing and typos.

- *Table Look-Ups*. If there are complex calculations, such as trigonometric functions, in time-critical portions of your code, consider precalculating all the possible results, placing them in an array, and simply retrieving the appropriate value at runtime to save calculating overhead. Not all complex functions lend themselves well to table look-ups — square roots, for instance, are difficult to do this way — but trig functions can be sped up a hundredfold through precalculation.

- *Fixed Point Arithmetic*. The standard floating point functions that came with your compiler can sometimes be quite slow, especially if there's no math coprocessor in the user's computer. Fixed point arithmetic, which handles fractional values using standard integer operations, can be much faster. Note, however, that this may not be true much longer, as Pentium-based computers (which come with extremely fast built-in math coprocessors) become increasingly common on gameplayers' desktops.

Finally, remember that knowing where to optimize is as important as knowing how to optimize. The secret? Look for deeply nested loops and put your optimization resources inside them. A code profiler, such as the Turbo Profiler that comes with Borland C++, can also be useful in locating performance bottlenecks.



13

Putting It All Together

13

e now have a nearly complete ray-casting engine, capable of drawing column by column the floor, ceiling, and walls of an environment described in the computer's memory as a series of two-dimensional arrays. What are we going to do with this engine? Well, we could use it to create a virtual reality program in which users walk through an imaginary environment defined by its floors, ceilings, and walls. And that's essentially what we're going to do, except that this imaginary environment will be used in the larger context of a computer game. While we won't present a complete game here, we'll give you all the tools that you'll need to design your own. And we'll create a kind of virtual reality application to demonstrate those tools.

In this chapter, we'll look at ways in which our ray-casting engine can be used to generate graphics for a computer game. We'll need to add one more feature to our engine before such games become possible. That feature will add objects, both moving and stationary, to our imaginary world. This will allow us in turn to populate a maze with objects such as treasures (which the player can collect for additional points) and monsters (which the player must destroy in order to complete his or her exploration of the maze).

The problem with adding objects to our maze is that the code required to add such objects necessarily constitutes a kludge. *Kludge* (pronounced "klooge") is a term popularized by progammers in the post World War II era to describe

program features that are neither elegant nor simple to code. Kludges make a program complex and, well, ugly, but they are sometimes necessary to make the program run well. Up until now, our ray-casting engine has been remarkably simple and elegant. But now we have to recognize the requirements of real-world game programming and add some rather awkward features to our engine.

Adding Objects to the World

There are several methods that we could use to add objects to our "world." For instance, we could create a two-dimensional "object map" array, similar to the wall, floor, and ceiling arrays that we've used up until now, that would indicate where each object is located within the maze. This would allow us to identify the location of objects during the wallcasting portion of the ray-casting process. Each object that we find while casting for the nearest wall could be pushed onto a stack — a data structure from which objects are removed in the opposite order from that in which they were recorded, or "pushed," onto the structure — and then removed from the stack in reverse order to be drawn on the display. However, this would greatly slow down the ray-casting process, since each object would be encountered by *several* rays — one for each column in the width of the displayed object — and would therefore be processed in a thoroughly redundant manner, once for every ray that struck the object. In fact, this is how we draw walls in our current ray-casting system. But drawing objects using the same method might take up too much time for our purposes.

A better way to identify objects would be to create a "visibility array," with one element for each square in the maze. All elements of this array could be set to 0 before ray casting begins. Then, during the ray-casting process, we can note which maze squares are visible by placing non-zero values in the appropriate array elements. When ray casting is complete, we can check this array to see which squares are visible and consult an "object array" to determine if those squares contain objects. If they do, those objects can be drawn by a special object-drawing function. The problem with this method is that the visible squares may not be *completely* visible; that is, portions of these squares may be obscured by walls that lie closer to the viewer, so that the objects in these squares are only partly visible. How does the object-drawing function know whether it is to draw an entire object or only part of an object?

Hidden Surface Removal

By and large, this second method provides the more efficient way to draw objects (though don't let this stop you from searching for more efficient methods of

rendering objects using other algorithms). Using this method, we can search for visible squares during the ray-casting process, recording the identity of visible squares in an array that we'll set up explicitly for that purpose, then draw the objects once the walls, floors, and ceiling are drawn.

Unfortunately, this algorithm opens a Pandora's box of problems, which can collectively be referred to as *hidden surface removal* problems. We've mentioned these problems earlier, but we haven't really tackled them. In the polygon-fill graphics that I discuss in *Flights of Fantasy,* I treat this problem in some detail; but in ray casting it rarely comes up. One of the major advantages of the ray-casting process is that the programmer never has to worry about whether one surface is in front of another, since the ray-casting process invariably identifies those surfaces that are closest to the viewer. For instance, if one maze block is partially obscured by another maze block, the rays cast from the viewer's position will inevitably strike the object that is visible through the current column of the viewport. No specific code for hidden surface removal is necessary.

But when we populate the maze with objects, mobile or otherwise, hidden surface removal immediately rears its ugly head. If we were dealing with polygon graphics, there would be no elegant solution to this problem (though a number of inelegant solutions are discussed in *Flights of Fantasy*). Fortunately, ray casting lends itself to some relatively simple solutions to the hidden surface problem. Because we are only dealing with two-and-a-half-dimensional graphics — that is, graphics in which only the x and y maze positions are variable and in which the vertical positions of all objects are fixed — we can come up with simpler solutions to the hidden surface problem than if we were dealing with true three-dimensional objects of the sort considered in *Flights of Fantasy.*

When we cast a ray through a column in the viewport, we learn the distance of the nearest wall visible through that column. If there are objects nearer to the viewer than that wall, then they will be visible and should be drawn on top of the wall in that column. If there are objects *farther* from the viewer than that wall, then they are not visible through that column and should not be drawn (see Figure 13-1).

Although this is a simple enough way to regard the hidden surface removal problem, it still treats those surfaces on a column-by-column basis. A given object may be visible through column A, but the same object may be blocked by a wall in column B (see Figure 13-2). As noted earlier, we could treat the hidden surface removal problem on a column-by-column basis, but the resulting code would be unnecessarily slow. We can draw objects much faster if we draw the objects all at once, after the ray casting is done. But how can we then determine which columns in the bitmapped images of those objects are visible and which are not?

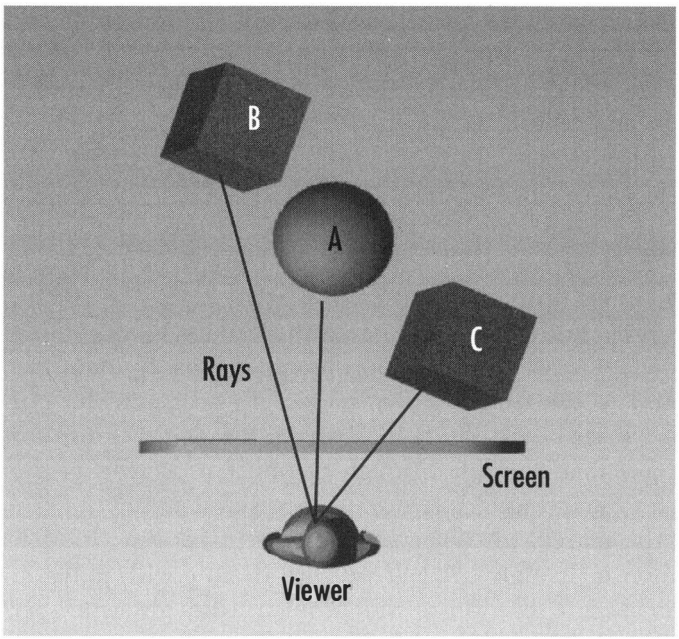


Figure 13-1 Wall B is farther from the viewer than object A, but wall C is closer

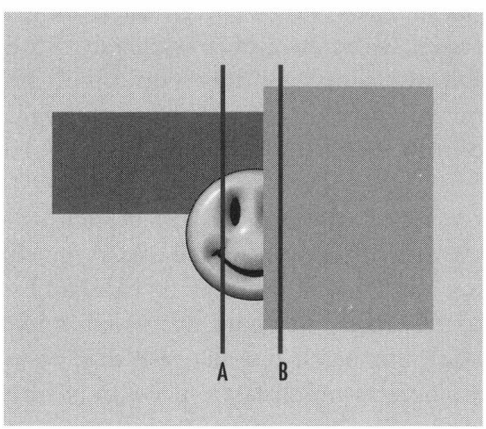


Figure 13-2 The object is visible through viewport column A, but not through column B

Column-by-Column Drawing

We can reach a fairly agreeable compromise between drawing an object a column at a time and drawing objects all at once: That solution is to record the hidden surface removal information while we draw the individual columns of the wall through wallcasting, then use that information to determine which columns of an object are visible and which are not during the drawing of each object.

As confusing as that may sound, it's not hard to implement in computer code. In mode 13h, there can be at most 320 columns of pixels in the viewport. When we cast rays through each of these pixel columns to determine the distance of the nearest wall column, it's a simple matter to record the distances to these wall columns in a 320-element array, in which each column represents a column of the viewport. We can then use this information to determine which surfaces are hidden and which surfaces are visible when drawing objects.

Thus we can create an array with 320 elements called dist[] that will contain the distance of the nearest wall, viewed from the current viewing angle, for each column in the viewport. (As a practical matter, we'll maintain an element for every column on the mode 13h screen, but only those in the viewport will actually be used. This will allow us to resize the viewport dynamically, as requested by the user.)

The distance[] Array

The distance[] array will be created in the *draw_maze()* function, in the OPTRAY.CPP module. Every time a wall is detected while casting a ray through a column in the viewport, the distance of that wall will be recorded in the element of array distance corresponding to that column of the display. For instance, if the ray cast through column 200 on the display strikes a wall at a distance of 389 floor pixels from the screen, array element dist[200] will be set equal to 389. This is simple to add to the *draw_maze()* function, like this:

```
dist[column]=distance;
```

where column is the column through which the most recent ray was cast. This line appears in the code three lines after the ray casting is finished. See the module FINALRAY.CPP in the GARDENS directory on the disk.

To determine which objects are (potentially) visible in the current viewport, we'll create an array called visible[][], where each element will correspond to a square in the current maze. The two elements in brackets following the visible array name will correspond to the x,y maze coordinates of the square through which a ray is being cast. For example, if the square at maze coordinates 7,4 is visible, then the array element visible[7][4] will be set equal to a value of -1, the

standard notation for "true." Later (as you'll see in a moment) we'll check for this value to see if the maze square at 7,4 (or any other location) is visible. The code for setting this element of the visible array to -1 will also be placed in the *draw_maze()* function as represented in the FINALRAY.CPP module. As each maze square is detected during the wallcasting process, the equivalent element of the visible array will be set to -1.

When do we actually draw the objects? We could do so immediately after the wallcasting process is complete, but the floorcasting and ceilingcasting processes would then overwrite portions of the object images. It would be better to wait until all of these processes are complete. That way, the images of the objects will sit neatly on top of the floor and ceiling images. Although this involves some redundant pixel drawing — we'll be erasing floor and ceiling (not to mention wall) pixels by overdrawing them with the pixels in the objects — it's actually easier to draw the objects over the walls than to draw only those portions of the floors, walls, and ceilings that will not need to be overdrawn by objects.

(Note that what's easier about this process is the programming itself. The reader is encouraged to come up with algorithms that will prevent drawing portions of the walls, floors, and ceilings that will later be overdrawn by objects from being drawn in the first place. This may make the code more complicated — not to mention harder to understand — but it could save time in the drawing itself and increase the frame rate of the resulting animation.)

Thus we'll place the object-drawing code at the very end of the *draw_maze()* function. Because we've recorded the distance information for the individual columns on the display, the function that actually draws the objects — which we'll call *objdraw()* — will do the hidden surface removal. In the *draw_maze()* function, we need only worry about where the object is to be drawn on the display. We'll then call *objdraw()* to perform the actual drawing.

Representing Objects

Before we can draw the objects, however, we'll need a means of representing them. First, as noted above, we'll create an object array, which will be initialized in the main module of our game, GARDENS.CPP, like this:

```
map_type objmap={
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

```
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};
```

Each element in this array corresponds to a square in the maze, according to the same system used in the wall, ceiling, and floor arrays that we developed in earlier chapters. If an element is set to 0, there is no object in that square of the maze. If it's non-zero, there is. For instance, if an element is set to 1, then we know that object 1 resides in that square.

How do we know what object 1 is? We'll create a separate array of structures to hold that information. The structures in the array will be of type *objtype*, which will be defined in the OBJDRAW.H file like this:

```
struct objtype {
  int screenx,screeny;
  int mazex,mazey;
  int wbit,hbit;
  int wdraw,hdraw;
  unsigned int distance;
  int imagenum;
};
```

The screenx,screeny fields hold the screen coordinates of the upper-left corner of this object as it will appear on the video display. This value will be calculated in the course of the code that follows. The mazex,mazey fields, which indicate the location of the object within the maze, will be filled in as the object is initialized in the GARDENS.CPP module. (I'll show you this code in a moment.) The wbit,hbit fields represent the width and height of the bitmapped image of the object, as it's stored in memory. (For all of the objects that we'll use, these will both be equal to 64, making the width and height of the object bitmaps the same as for the bitmaps we're using for walls, floors, and ceilings.) The wdraw and hdraw fields are the width and height of the bitmaps for the objects as they'll actually be drawn on the display. These figures will be calculated just before the objects are drawn and will be subject to clipping against the sides of the viewport. The distance field, which contains the distance from the viewer to the object, will also be calculated just prior to drawing. And the imagenum field corresponds to the index within an array of bitmapped object images that will be used to draw the objects on the display.

We can then initialize an array of these objects at the beginning of the GARDENS.CPP module:

```
objtype obj[]={
  {0,0,5*64,5*64,64,64,0,0,0,0,0},   // Skull
  {0,0,3*64+32,3*64+32,64,64,0,0,0,0,3},   // Column
};
```

The first line of figures represents the state of the "skull" object. Some of these fields will be calculated later, as noted above. These are given an initial value of 0. However, the object's mazex and mazey fields are both initialized to 5*64, which will put the object in the maze square with coordinates 5,5. (We multiply this by 64 because these coordinates are in the fine coordinate system of the maze rather than the maze square coordinate system.) The two fields set to 64 represent the dimensions of the bitmap associated with the object. The final number is the number of the element in the bitmap array that contains the bitmapped image of the object. We'll now create that array.

We'll set aside memory for the array that will hold the bitmapped images of the objects, like this:

```
char far *image[NUMIMAGES];
```

This creates an array of pointers to type char called image.

The images of the objects will be stored on disk in a PCX file called OBJS.PCX. We can load it using the PCX loader that we developed earlier in this book. However, instead of leaving the PCX in its image buffer and extracting the images directly from the buffer to the display, we'll copy each 64 by 64 image into its own 4096-byte buffer. We'll then set the individual elements of the image array to point to those buffers, so that we can retrieve the images later. This saves us some memory and simplifies the process of calculating the position of the image.

Here are the instructions that read the PCX file and copy the image data into the individual image buffers:

```
if (loadPCX("objs.pcx",&objs)) exit(1);

for (unsigned int i=0; i<NUMIMAGES; i++)
{
  if ((image[i]=new unsigned char[4096])==NULL)
      exit(1);
  for (unsigned int j=0; j<64; j++) {
    for (unsigned int k=0; k<64; k++) {
      image[i][k*64+j]=objs.image[((i/5)*320*64+(i%5)
                                    *64+j+k*320)];
    }
  }
}

delete(logo.image);
```

These instructions are in the GARDENS.CPP module. The NUMIMAGES constant is defined in the GARDENS.H header file and represents the number of 64 by 64 object images that are present in the OBJS.PCX file. The for() loop iterates once for each of these images, first allotting a 4096-byte buffer in which to store the image (and pointing the equivalent member of the images array to that buffer), then looping through the pixel data for the image and copying that data into an element of the image array, where the image data will be stored for the rest of the program. The delete statement returns the 64-kilobyte buffer used by the PCX file back to DOS, so that we can reuse it later for additional PCX images that we'll be loading on a temporary basis. (More about these images in a moment.)

Now we can begin actually drawing the objects.

Drawing the Objects

We'll walk through the object-drawing function itself in a moment. First, let's take a look at the code that calls that function. The code that calls the object drawing function will be placed at the very end of the *draw_maze()* function. It will first iterate through the visible() array to determine which squares in the maze can actually be seen:

```
for (xsquare=0; xsquare<16; xsquare++) {
  for (ysquare=0; ysquare<16; ysquare++) {
    if (visible[xsquare][ysquare]) {
```

If a square is visible, we'll proceed to determine if it contains an object and, if so, where on the video display the object should be drawn. First, we'll get the number of the object from the objmap[] array that we established earlier:

```
int objnum=objmap[xsquare][ysquare];
```

If the object number in the specified array element is 0, then there's no object there. If it's *not* 0, we'll continue to process the object that we've found there:

```
if (objnum) {  // If object
```

The image-drawing code, having determined that a 0 is present in one of the visible squares, now prepares to draw that object. First, the object number must be decremented by one:

```
--objnum;
```

Why must we decrement the object number? For the same reason that we decrement the tile number for a wall after it has been extracted from the wall array. We identify the presence of objects in the object map by looking for values that aren't 0. This means that there can't be any 0 objects in the map. However,

there *is* a 0 object in the object array. Thus, we must represent 0 objects in the object map as object 1. This in turn pushes all other object numbers up by a count of 1. Therefore, when extracting these numbers from the array, we must decrement them to compensate.

To simplify subsequent code, we'll create a pair of integer variables called *x* and *y* to hold the x,y coordinates of the object within the maze grid:

```
int x=obj[objnum].mazex-xview;
int y=obj[objnum].mazey-yview;
```

Now comes the trickiest part of the object-drawing code. In order to determine the screen coordinates at which the object must be drawn, we must rotate the object around the viewer until it is aligned with the viewer's coordinate position. This subject was discussed in some detail back in chapter 7. To align an object with the current viewing position, we must rotate it around the viewer by the *negative* of the viewer's current rotational position. (Readers who find this concept a bit hazy are urged to read *Flights of Fantasy*, where the subject of aligning objects with the viewer position was discussed in great detail.) We can perform this rotation with these instructions:

```
int alignx=(x*COS(-viewing_angle)
  - y*SIN(-viewing_angle))>>16;
int aligny=(x*SIN(-viewing_angle)
  + y*COS(-viewing_angle))>>16;
```

You'll recall from chapter 7 that these equations rotate an object around an origin position (that is, the 0,0 point relative to which the object coordinates are defined). All of the objects in our maze are defined by a two-dimensional coordinate position, so this is all we need do to adjust their coordinates relative to the current viewing position. (If we were dealing with three-dimensional, rather than two-and-a-half-dimensional, graphics, we would also need to rotate the coordinates around the y and z axes of our coordinate system.) The variables *alignx* and *aligny* tell us where the object is, or would be, if the viewer were rotated to angle 0 — directly facing north along the y axis — within the coordinate system.

Since this operation realigns the objects as though we were facing northward along the positive y axis, we can tell if the object is behind us or in front of us by checking to see if its y coordinate is positive or negative (see Figure 13-3). If it's positive, the object is in front of us. If it's negative, the object is behind us. Thus we will only proceed with our drawing code if the object's y coordinate is positive, that is, if the object is in front of us:

```
if (aligny>=0) {
```

Of course, this problem should have been solved by the earlier if statement that only accepted squares marked as visible, so this code is a bit redundant. It's

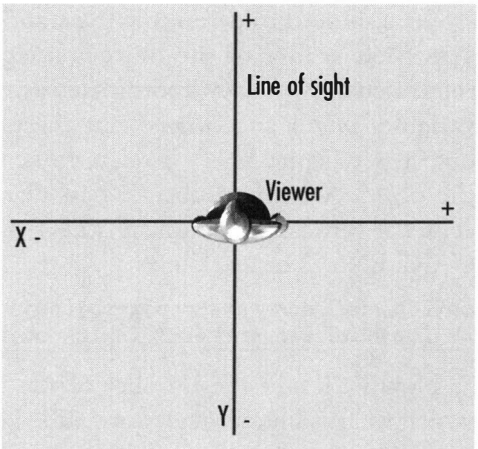


Figure 13-3 When facing northward along the positive y axis, the value of the y coordinate tells us if an object is in front of us (positive coordinate) or behind us (negative coordinate)

probably possible to remove the visibility marking code without penalty, though this is left as an excercise for the reader.

We'll use the familiar Pythagorean method to determine how far away each object is:

```
distance=sqrt((long)alignx*alignx+(long)aligny*aligny);
```

This assigns the distance of the object, in maze pixels, to the integer variable *distance* that we defined earlier in the *draw_maze()* function.

We can now project the object's position onto the video display using a standard perspective equation. Interestingly, it's only necessary to project the object's x coordinate onto the display, since this is the only position that's actually variable within our two-and-a-half-dimensional coordinate system:

```
int projx=alignx*VIEWER_DISTANCE/distance+HORIZ_CENTER;
```

Now we need to determine the horizontal and vertical size of the object as it will be displayed. We can do this with the standard perspective equations too:

```
obj[objnum].wdraw=obj[objnum].wbit*VIEWER_DISTANCE/
  distance;
obj[objnum].hdraw=obj[objnum].hbit*VIEWER_DISTANCE/
  distance;
```

This code sets the wdraw and hdraw fields of the object structure, which represent the width and height of the object on the display, to the appropriate perspective-adjusted values.

We also need to determine the x and y coordinates on the display at which the upper-left corner of the object's image should be drawn. A moment ago, we calculated the x and y coordinates at which the object will be displayed, in the variables *alignx* and *aligny*. But these are the coordinates of the center of the object (see Figure 13-4). To determine the coordinates of the upper-left corner of the object, we must subtract half of its width and height from this value and center it vertically around the VERT_CENTER of the display (as defined in the GARDENS.H header file):

```
obj[objnum].screenx=-projx-obj[objnum].wdraw/2;
obj[objnum].screeny=VERT_CENTER-obj[objnum].hdraw/2;
```

Now we'll add the number of the object to the list of objects to be drawn, which we'll maintain in an array called objlist:

```
objlist[numobjs++]=objnum;
```

The *numobjs* variable was established back before we began looping through the objects, like this:

```
int numobjs=0;
```

By incrementing the value of *numobjs* every time an object number is added to objlist, we know how many objects are currently visible.

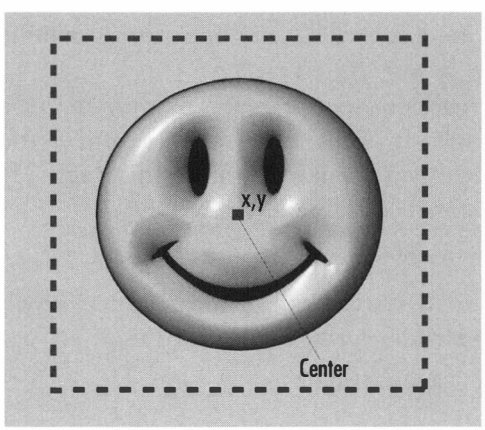


Figure 13-4 The x,y coordinates mark the center of the object. Before we draw it, we must calculate the coordinates of its upper-left corner on the display

The Painter's Algorithm

We're done compiling the objects list. (I'll spare you all of the closed curly brackets that are necessary to terminate the loops that we started in this process.) Now we can start a new loop that actually draws the objects.

First, however, we need to perform some more hidden surface removal. Earlier, we developed a method for detecting whether a given column of an object's bitmap is behind or in front of a wall, for any given column of the viewport. However, we also need to know if one object is in front of another object. To determine this, we'll use a technique called the Painter's Algorithm.

The Painter's Algorithm gets its name from the idea that painters, when rendering a landscape, paint background objects first, then paint foreground objects over the background objects. Translated into computer rendering terms, the Painter's Algorithm says that we should draw objects in the background before we draw objects in the foreground, so that objects in the foreground will be drawn on top of background objects.

This requires that we perform an operation known as a *depth sort*. A depth sort takes all of the objects that are visible (or potentially visible) on the display and sorts them into a list with the objects most distant from the viewer at the beginning of the list and the objects closest to the viewer at the end of the list. If we then draw the objects in this order, it is guaranteed that foreground objects will be drawn on top of background objects.

In a polygon graphics system of the type that is used in most flight simulators, this depth sort may sort objects into the wrong order. This is because polygons can be rotated at odd angles to the viewer, allowing background objects to obscure foreground objects. The objects that we'll be drawing in this chapter, however, will *not* be oriented at such odd angles. Rather, each object will face squarely toward the viewer, with all pixels in the object at the same distance from the display (see Figure 13-5). Thus a simple depth sort is all we need in order to determine a correct drawing order for these objects.

We'll use the simplest form of sort, the bubble sort, to arrange the objects in the proper front-to-back order:

```
if (numobjs) {
  int swap=-1;
  while (swap) {
    swap=0;
    for (int i=0; i<(numobjs-1); i++) {
      if (obj[objlist[i]].distance<obj[objlist[i+1]].distance) {
        int temp=objlist[i];
        objlist[i]=objlist[i+1];
        objlist[i+1]=temp;
        swap=-1;
      }  // End if
```

```
      }  // End for
    }  // End while
  }  // End if
```

This sorting algorithm iterates through the list of objects, swapping pairs of objects into the correct order when it finds a pair in which the closer object is lower in the list than a more distant object. When all of the swaps are completed (as indicated by the *swap* variable, which is set to -1 when a swap occurs and to 0 when one has not occurred), the sort terminates.

For small numbers of objects, the bubble sort does a fine job. For larger numbers of objects, the reader is encouraged to find a faster swapping method — check any book of computer algorithms for suggestions — and to substitute it for the bubble sort. One possibility is the insertion sort, in which the objects are inserted in order as the list is compiled, without a separate sorting step. This particular sort would be most efficiently performed with a linked list, in which each entry has a pointer to the next item, than as an ordered list of the sort we're using here, since inserting an item into a linked list is a relatively trivial operation.

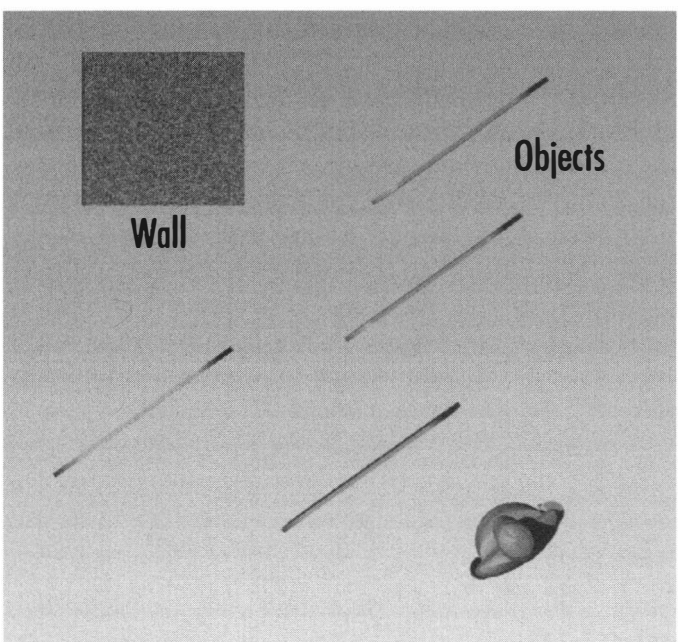


Figure 13-5 All of the objects will be facing squarely toward the viewer, no matter what the viewer's position

Now that the objects are sorted, we can draw them. The objlist array contains the numbers of the object in the order in which they should be drawn; we need only iterate through this array:

```
  for (int i=0; i<numobjs; i++) {
    drawobject(&(obj[objlist[i]]),VIEWPORT_LEFT,VIEWPORT_TOP,
       VIEWPORT_RIGHT,VIEWPORT_BOT,distance,screen);
  } // End for
}
```

This terminates the *draw_maze* function. You'll notice that the final line of the function calls a new function that we've named *drawobject()*, which (as its name implies) does the actual object drawing. Next, we'll look at how this function is defined.

The drawobject() Function

The prototype for the *drawobject()* function looks like this:

```
void drawobject(objtype *obj,int lclipx,int lclipy,int rclipx,
                int rclipy,unsigned int distance,char *screen);
```

This prototype, which you can find in the OBJDRAW.H file — the function itself is in the OBJDRAW.CPP file — tells us that the *drawobject()* function requires these parameters: *obj, which is a pointer to the objtype structure that defines the object; lclipx, which is the x coordinate of the left edge of the viewport; lclipy, which is the y coordinate of the left edge of the viewport; rclipx, which is the x coordinate of the right edge of the viewport; rclipy, which is the y coordinate of the right edge of the viewport; distance, which is the distance (in floor pixels) of the object from the viewer; and *screen, which is a pointer to the current video display or video display buffer.

The *drawobject()* function first takes the pointer to the bitmap representing the object and assigns it to the char pointer bitmap:

```
char far *bitmap=image[obj->imagenum];
```

Next, we determine the scale at which the object is to be drawn in the horizontal and vertical dimensions and assign these fractional values to the variables *fix_wincrement* and *fix_hincrement*, just as we did in the last chapter:

```
long fix_wincrement=(float)obj->wbit/obj->wdraw*SHIFT_MULT;
long fix_hincrement=(float)obj->hbit/obj->hdraw*SHIFT_MULT;
```

These variables, as their names imply, contain fixed point values, that is, fractions represented as *long integer*s. Now we need to determine the position in video memory (buffered or otherwise) at which we should begin drawing the object:

```
unsigned int screenstart=obj->screeny*320+obj->screenx;
```

The variable *screenstart* now contains the video memory offset at which drawing should begin.

We'll need a variable to indicate which vertical column of the object's image we're currently drawing. This should be initialized to zero:

```
unsigned long column=0;
```

The reason that we've used a *long* variable here is that *column* will be treated by subsequent code as a fixed point variable.

It would also be useful if we knew how many pixels (that is, bytes) the *screenstart* variable is indented from the left side of the display:

```
unsigned int leftrow=screenstart % 320;
```

We'll be drawing the object as a series of columns from left to right. Let's use a for() loop to iterate through the columns:

```
for (unsigned int x=0; x<obj->hdraw; x++) {
```

The *screenstart* variable tells us where in video memory we should begin drawing the object. However, we'll set the variable *sptr* equal to the offset in video memory where the next pixel should be drawn:

```
unsigned int sptr=screenstart++;
```

The variable *bptr* will be set equal to the column in video memory at which the next pixel is to be displayed. Since the variable *column* is a fixed point variable, we'll need to shift the value of *column* 16 positions to the left to represent the integer value that we're storing in *bptr:*

```
unsigned int bptr=column>>16;
```

To advance the variable *column* to its next position, we'll need to add the value of *fix_wincrement* to it:

```
column+=fix_wincrement;
```

Now, before we begin drawing the object, we need to create two new variables, which we'll call *line* and *lastline:*

```
unsigned long line=0;
unsigned int lastline=0;
```

Now we can set the *unsigned integer* variable *column* to point to the upper-left corner of the object, by adding *x* and *leftrow:*

```
unsigned int column=x+leftrow;
```

We'll draw the object by iterating from the left clipping parameter to the right clipping parameter:

```
if (((column)>lclipx) && ((column)<rclipx)) {
```

Here the hidden surface removal comes into play. We should not draw the current column at all if there's a wall closer to the viewer than the object:

```
if (distance<dist[column]) {
```

If the wall is farther from the viewer than the object, we'll draw the object in front of it. To do this, we'll use a for() loop that will iterate through all of the columns that are visible in the object:

```
  for (unsigned int y=0; y<obj->wdraw; y++) {
    int b=bitmap[bptr];
    if (b) screen[sptr]=b;
    line+=fix_hincrement;
    if (lastline!=(line>>16)) {
      bptr+=obj->wbit;
      lastline=line>>16;
    }
    sptr+=320;
  }
}
```

The most important line here is the if() statement if (b) screen[sptr=b;. This line checks to see if a pixel is 0 before printing it to the display. Why? Because 0 pixels in an object image represent transparency. Zero pixels should not be displayed; rather, the background pixels behind these pixels should show through. If not for these transparent pixels, all objects would be perfectly rectangular, which is obviously undesirable. Instead, we allow 0 pixels to be transparent. When we detect such a pixel, we don't draw over the background, allowing the background to show through.

And that, aside from the closing brackets, is all that there is to drawing objects.

Adding Game Features

Learning to program isn't all that there is to creating a game. You also have to give some thought — preferably a great deal of thought — to game design. Alas, game design is at least as complex a subject as game programming, and really deserves a book of its own. We can, however, offer a few suggestions to help you get started.

Start by looking at games that you've enjoyed playing. Try to identify the features that make those games attractive: graphics, atmosphere, story line, and so on. Decide what genre of game you'd like to create. For example, an adventure game emphasizes exploration and puzzle-solving. In such a game, the player needs to acquire the objects necessary to solve a puzzle, and perhaps select actions

to be performed on those objects — lift the rock, push the button, open the drawer, and so on. In a role-playing game, on the other hand, the player takes the part of a character who grows in experience and power while overcoming monsters and fulfilling quests. Many successful games (such as the Ultima series) combine aspects of both adventure and role-playing.

Obviously, any game designed around the algorithms in this book will take place in some kind of maze. Once you've decided what kind of game you want to create, think about the kinds of objects to be found in that maze. In addition to creating pictures for the objects, you'll need to expand the objtype structure to account for the important characteristics of objects in your game. For example, you might add structure members like these:

```
char * name     // name of object
int owner;      //  0 = unowned, 1 = owned by player
                // 2-99 owned by monster
#int weight     // how much object weights
                // (player can only carry his/her
                // strength in objects)
int type        // 1 = weapon 2 = treasure 3 = food
                // 4-99 = special object
```

You could then use auxiliary structures for information specific to particular types of objects. For example, a weapon would do a certain amount of damage, food might restore a certain amount of the player's strength, and so on.

You will also need a structure to hold information about the player as represented by the "viewpoint character" in the game. Some traditional characteristics used in role-playing games include:

```
int str     // how strong: affects damage done in attacks,
            // ability to carry objects
            // force doors open, and so on
int dex     // how fast/agile the player is;
            // ability to dodge attacks or traps
int con     // constitution, or ability to absorb damage
int hp      // hit points (amount of damage player can take
            // before dying)
int exp     // experience points (given for accomplishments,
            // increases abilities of character)
```

You will probably need a structure to hold similar information about monsters, though it probably won't be as detailed as the structure for the player. Once you've built this "infrastructure" of structures and arrays, you'll need a main "command processor" function that interprets the player's mouse clicks and sends control to the appropriate function. Typical player actions might include:

- Move (click on direction arrows or use keyboard)

- Turn around
- Examine an object
- Pick up an object
- Attack a monster
- Review information about the player character
- Save the game

and so on. It's up to you whether to use a menu interface or an icon interface, but the latter is more popular in contemporary games. Of course, we've already provided you with mouse, keyboard, and joystick input routines.

Systems for manipulating objects (and particularly combat and magic systems) can become very complex. We recommend you start with simple functions and add features only after the game has been tested and is working well.

Appendix A lists some further readings on game programming and design that you may find helpful.

Bells and Whistles

Before I'm done, however, I'd like to add a few additional bells and whistles to our "game." Specifically, I'd like to add a set of opening credits and an automap.

First, let's add the opening credits.

The Opening Credits

The opening credits will be a series of PCX files displayed one after another. The code for displaying each PCX file will be identical to the code for displaying the rest. Here's the code for displaying the first credit:

```
fadepalout(0,256);

  // Load Waite Group logo:
  if (loadPCX("waite.pcx",&logo)) exit(1);

  // Copy logo into video memory:
  for (i=0; i<64000; i++)
    screen[i]=logo.image[i];

  // Fade logo in:
  fadepalin(0,256,logo.palette);
```

```
// Start timer running:
timer1.timerOn();

elapsed_time = timer1.getElapsed();

while(elapsed_time<TITLE_DELAY) {
  if (kbhit()) {
    abort_titles=-1;
    break;
  }
  elapsed_time = timer1.getElapsed();
}

fadepalout(0,256);
cls(screen);

delete(logo.image);

timer1.timerOff();
```

This code begins with an interesting instruction: fadepalout(0,256);. To what does this code refer? It refers to a special set of C++ functions stored on disk in the file PALFADE.CPP. (The header file defining these functions is available in the file PALFADE.H.) The code in this file was written by Mark Betz for *Flights of Fantasy*. The *palfadeout()* function fades all of the colors in the current 256-color palette to color 0,0,0 — that is, black. This causes whatever is on the screen when this program is run to fade to black. We then use the *loadPCX()* function to load the initial credit screen, contained on the disk in the file WAITE.PCX. This PCX image contains a three-dimensionally rendered version of The Waite Group logo, created by artist Stephen Blackmon using the rendering program Imagine 3.0 from Impulse Software. The for() loop that follows copies this bitmap data into video memory, then the fadepalin(0,256,logo.palette) instruction fades this image up on the video display.

We also use the *kbhit()* function to check if the user hits a key while the logo is on the display. This allows the user who has already seen the title sequence enough times to memorize individual pixels to abort the credits and proceed directly into the game. We also set the abort_titles flag to -1 so that we know to skip the succeeding credit screens.

If the user does *not* hit a key, we hold this image on the display for the amount of time defined in the TITLE_DELAY constant, using Mark Betz's timer routines (introduced in chapter 12):

```
elapsed_time = timer1.getElapsed();

while(elapsed_time<TITLE_DELAY) {
```

```
    if (kbhit()) {
     abort_titles=-1;
     break;
   }
   elapsed_time = timer1.getElapsed();
}
```

Then we fade out on these images and clear the display, using the functions in PALFADE.CPP:

```
fadepalout(0,256);
cls(screen);
```

Finally, we delete the memory necessary for storing the first title screen and turn the timer off:

```
delete(logo.image);

timer1.timerOff();
```

Then we do the same thing for two more title screens, the first of which contains an image of a mountain with the superimposed logo "A Chris Lampton Production" and the second the title of the game itself (see Figures 13-6a–c for the credit images). Note, however, that we now check to see if the abort_titles flag has been set before displaying the additional screens.

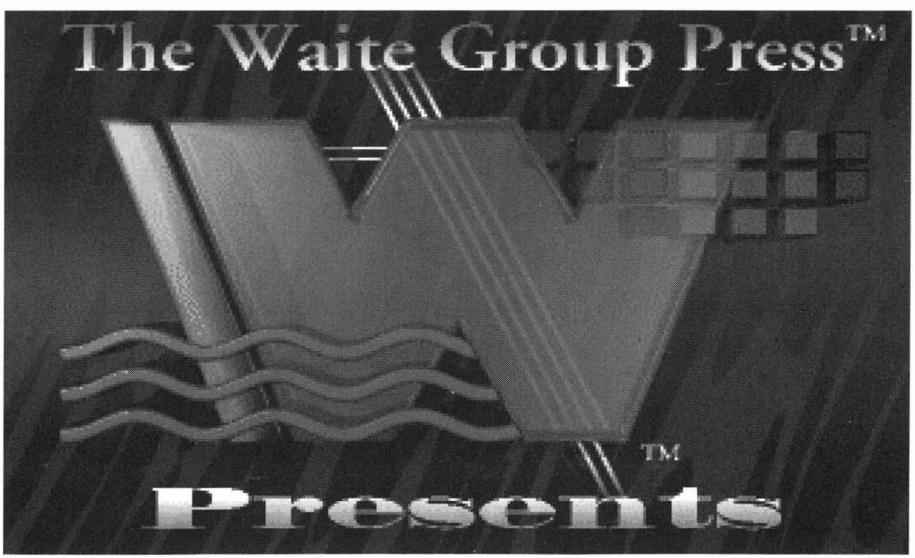


Figure 13-6a The Waite Group logo

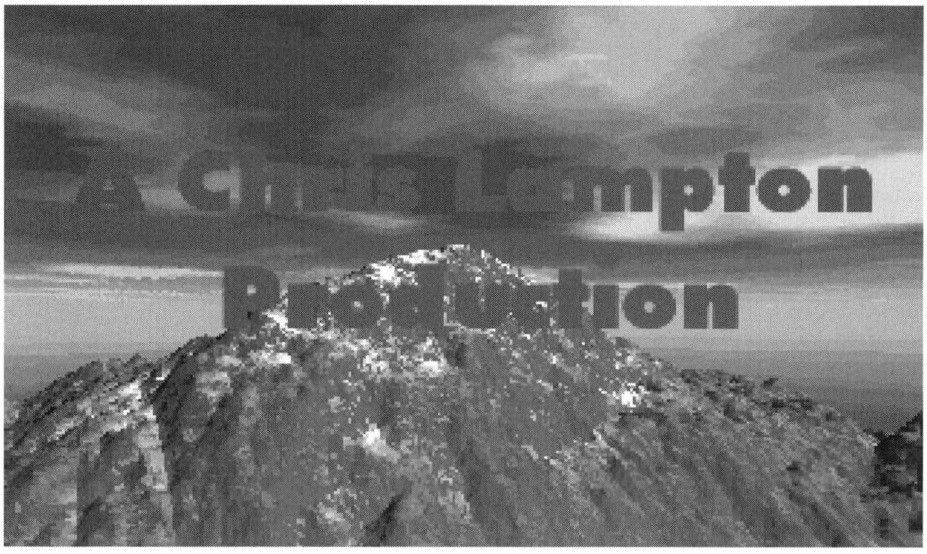


Figure 13-6b A Chris Lampton production, natch

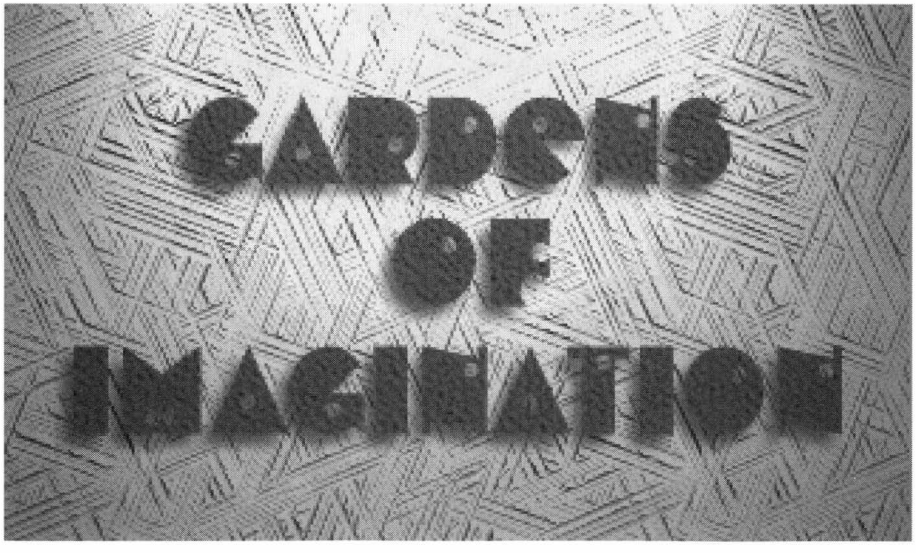


Figure 13-6c The name of the game

```
fadepalout(0,256);

  if (!abort_titles) {

    // Load mountain logo:
    if (loadPCX("chrispro.pcx",&logo)) exit(1);

    // Copy logo into video memory:
    for (i=0; i<64000; i++)
      screen[i]=logo.image[i];

    // Fade logo in:
    fadepalin(0,256,logo.palette);

    // Start timer running:
    timer1.timerOn();

    elapsed_time = timer1.getElapsed();

    while(elapsed_time<TITLE_DELAY) {
      if (kbhit()) {
        abort_titles=-1;
        break;
      }
      elapsed_time = timer1.getElapsed();
    }

    fadepalout(0,256);
    cls(screen);

    delete(logo.image);

    timer1.timerOff();

    fadepalout(0,256);
  }

  if (!abort_titles) {

    // Load title background:
    if (loadPCX("titlebg.pcx",&logo)) exit(1);

    // Copy logo into video memory:
    for (i=0; i<64000; i++)
      screen[i]=logo.image[i];

    // Fade logo in:
    fadepalin(0,256,logo.palette);

    // Start timer running:
    timer1.timerOn();
```

continued on next page

continued from previous page

```
    elapsed_time = timer1.getElapsed();

    while(elapsed_time<TITLE_DELAY) {
      if (kbhit()) {
        abort_titles=-1;
        break;
      }
      elapsed_time = timer1.getElapsed();
    }

    fadepalout(0,256);
    cls(screen);

    delete(logo.image);

    timer1.timerOff();
  }
```

The Automap

And that's *almost* all the code we need to complete our "game." However, there's one more element that I'd like to add: an automap.

What's an automap? The three-dimensional graphics produced by a maze game can often be confusing. The player can get lost in all of those three-dimensional twists and turns. An automap is a top-down map of the maze, in a format similar to the maps that you get from AAA or your local gas station, displaying where the walls are in the maze. We can create one quite easily by adding a bit of code to our GARDENS.CPP module. We'll call the automap-drawing function *drawmap*. Here's the code:

```
void drawmap(char far *screen)
{
  unsigned int ptr=MAPY*320+MAPX;
  for (int x=0; x<16; x++) {
    for (int y=0; y<16; y++) {
      if (walls[x][y]) {
        screen[ptr]=WALL_COLOR;
        screen[ptr+1]=WALL_COLOR;
        screen[ptr+2]=WALL_COLOR;
        screen[ptr+3]=WALL_COLOR;
        screen[ptr+320]=WALL_COLOR;
        screen[ptr+321]=WALL_COLOR;
        screen[ptr+322]=WALL_COLOR;
        screen[ptr+323]=WALL_COLOR;
        screen[ptr+640]=WALL_COLOR;
        screen[ptr+641]=WALL_COLOR;
        screen[ptr+642]=WALL_COLOR;
        screen[ptr+643]=WALL_COLOR;
        screen[ptr+960]=WALL_COLOR;
        screen[ptr+961]=WALL_COLOR;
        screen[ptr+962]=WALL_COLOR;
```

```
        screen[ptr+963]=WALL_COLOR;
      }
      ptr+=4;
    }
    ptr+=4*320-64;
  }
}
```

The *automap()* function requires only one parameter: *screen. This tells us the address of the video display or screen buffer. The constants MAPX and MAPY contain the x,y screen coordinates of the upper-left corner of the automap, which the function uses to calculate the video memory offset of the pixel in that corner, assigning that offset to the variable *ptr*. The two nested for loops iterate through the maze, drawing a white square (the color of which is defined in the constant WALL_COLOR) wherever it finds a wall. You'll notice that we've "unrolled the loop" (as discussed in chapter 12) when drawing the white square. Although it would be easier, and more compact, to draw the square with yet another pair of nested for loops (one to iterate across the square's width and one to iterate along its height), it's a bit faster simply to draw the wall square with a series of assignment statements placing the proper pixel values in all 16 of the video memory locations that constitute the 4 by 4-pixel square.

The position of the player within the map should also be depicted in the automap. This is performed by the function *drawplayer()*.

```
void drawplayer(char far *screen)
{
  unsigned int ptr=MAPY*320+xview/64*1280+MAPX+yview/64*4;
  screen[ptr]=PLAYER_COLOR;
  screen[ptr+1]=PLAYER_COLOR;
  screen[ptr+2]=PLAYER_COLOR;
  screen[ptr+3]=PLAYER_COLOR;
  screen[ptr+320]=PLAYER_COLOR;
  screen[ptr+321]=PLAYER_COLOR;
  screen[ptr+322]=PLAYER_COLOR;
  screen[ptr+323]=PLAYER_COLOR;
  screen[ptr+640]=PLAYER_COLOR;
  screen[ptr+641]=PLAYER_COLOR;
  screen[ptr+642]=PLAYER_COLOR;
  screen[ptr+643]=PLAYER_COLOR;
  screen[ptr+960]=PLAYER_COLOR;
  screen[ptr+961]=PLAYER_COLOR;
  screen[ptr+962]=PLAYER_COLOR;
  screen[ptr+963]=PLAYER_COLOR;
}
```

Once again, this function only needs to know where video memory (or the screen buffer) is located. It then draws a 4 by 4 square in the PLAYER_COLOR to represent the position of the player within the maze, using the same sort of unrolled loop that we used to draw the wall squares. The location of the player

square is calculated using the MAPX and MAPY constant, plus the *xview* and *yview* variables, which represent the current maze location of the player. (These variables are divided by 64 to determine which maze square the coordinates fall within.)

We need to add one more function before our automap is complete. This function, which is identical to the *drawplayer* function except in the choice of pixel color, will erase the image of the automapped player from the previous iteration of the animation:

```
void eraseplayer(char far *screen)
{
  unsigned int ptr=MAPY*320+xview/64*1280+MAPX+yview/64*4;
  screen[ptr]=0;
  screen[ptr+1]=0;
  screen[ptr+2]=0;
  screen[ptr+3]=0;
  screen[ptr+320]=0;
  screen[ptr+321]=0;
  screen[ptr+322]=0;
  screen[ptr+323]=0;
  screen[ptr+640]=0;
  screen[ptr+641]=0;
  screen[ptr+642]=0;
  screen[ptr+643]=0;
  screen[ptr+960]=0;
  screen[ptr+961]=0;
  screen[ptr+962]=0;
  screen[ptr+963]=0;
}
```

And that's all we need to do to create an automap. This code can then be called from the *main()* function.

Tracking the Player

In many games, the automap includes an additional feature. It shows where the player has been. Initially, the automap in such games is blank, but the squares of the map are gradually filled in as the player discovers them. This puts an emphasis on exploration, offering the player the exciting experience of "discovering" the maze as he or she wanders through it. It also gives the player guidance by pointing out which parts of the maze have not yet been explored. Without this kind of guidance, the player may spend much of the game wandering in circles, which can be a frustrating experience (and won't necessarily endear your game to the player).

How would you add such a feature to our demo? It wouldn't be difficult. Create a visibility array — that is, an array of integers in which each element corresponds to a maze square. (We've already set up several similar arrays in the

demo.) Initialize every element to 0. Then, as the player moves, set those elements corresponding to the square in which the player is standing, and the squares immediately adjoining it, to 1. When drawing the automap, only draw those squares that have been set to 1 in this array; all other squares should be black (or some other color indicating that they have not yet been explored).

Of course, this is not an entirely realistic method of representing the player's exploration of the maze. Most of us can see farther into the distance than the equivalent of a few maze squares. If you're feeling ambitious, you might want to show all squares through which the player's visual rays have been cast. This would require making the visibility array accessible to the ray-casting engine and marking all squares through which a ray has been cast as having been seen.

Playing the Game

That's it. We've created the framework for a ray-cast arcade game. To play the game, go to the directory called GAME on the distribution disk and type GARDENS. First you'll see the three title screens. Then we fade up on the main game screen, with a red border created using Fractal Design's Painter, an extremely useful program for generating incidental game graphics. (The title screen was also created in Painter.) The border is loaded in the module GARDENS.CPP, using the same code that loaded the background for the demo in the last chapter. (See Figure 13-7 for a look at the border.)

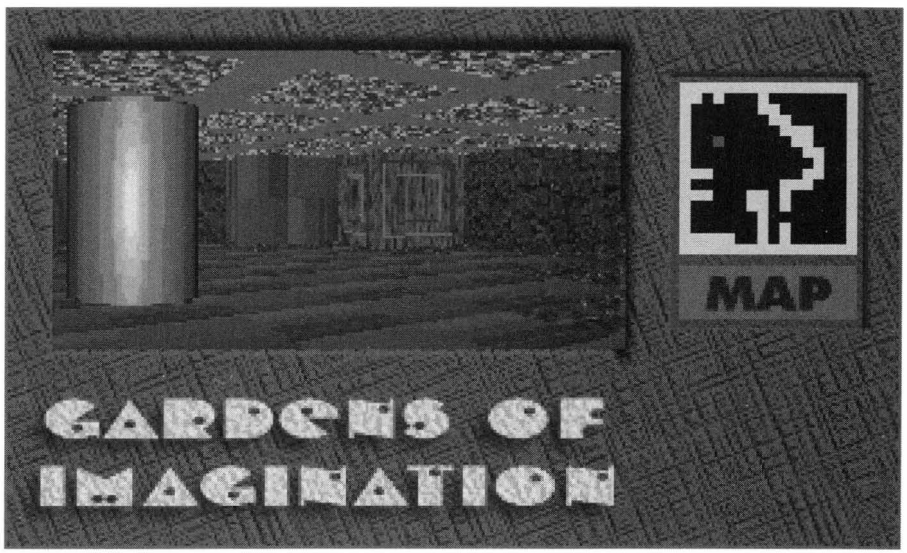
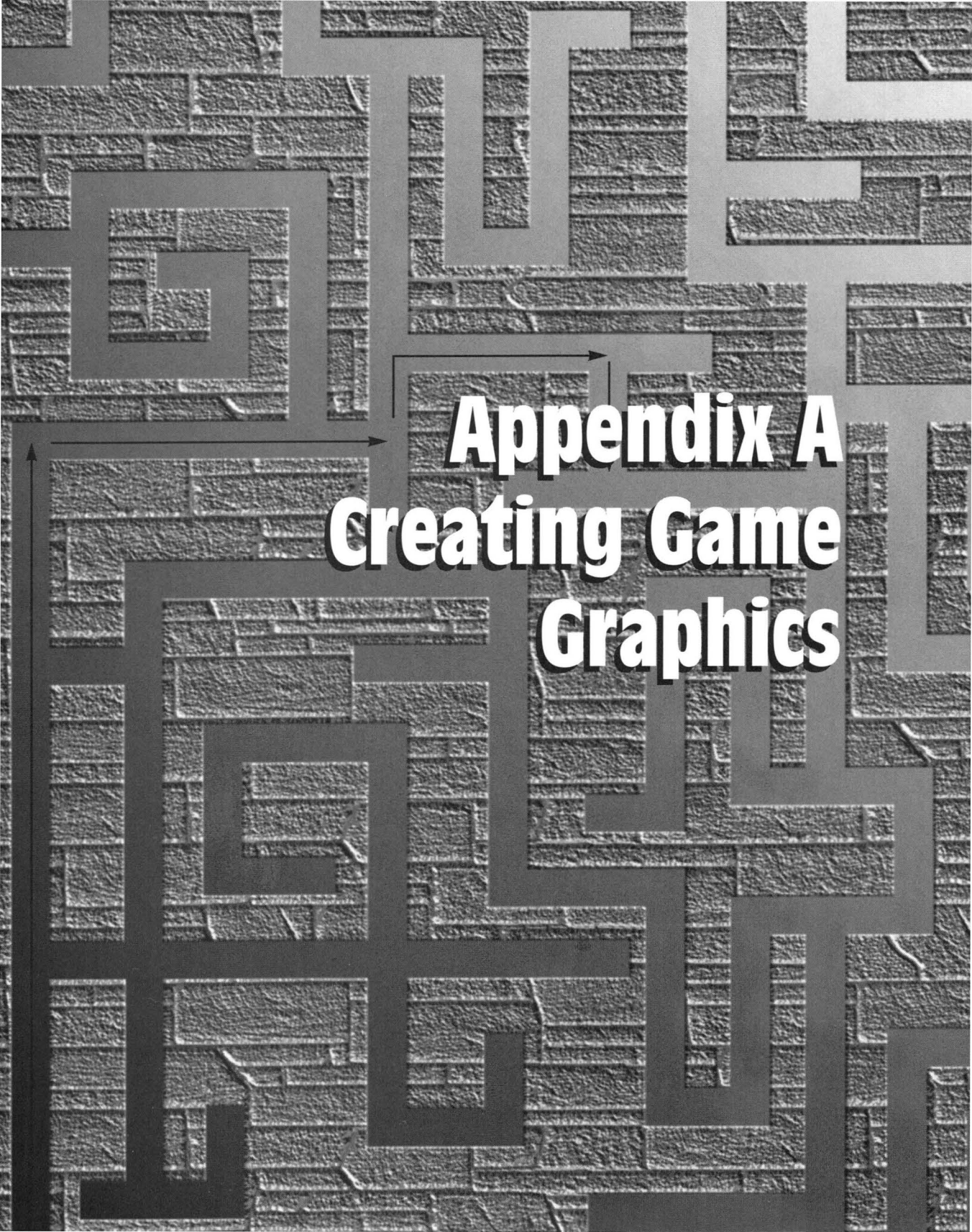


Figure 13-7 The main screen of program GARDENS.EXE

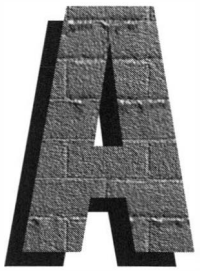
You can now move around through the maze, just as you did in the last chapter. The difference is that there are now objects in the maze, you can no longer walk through walls (because we check to see if the viewer is about to enter a maze square containing a wall block before allowing forward movement), the automap on the right side of the screen monitors your every movement, and there are objects at intervals throughout the maze, including cylindrical columns descending from ceiling to floor and floating skulls.

You are encouraged not only to *play* this game but to play *with* it: Expand it, improve it, and make it into an altogether viable commercial game. I've given you the tools to do so, now do it.

You may find that it's as much fun to create a maze game as it is to play one!



Appendix A Creating Game Graphics



For those of us whose artistic talents have never developed much past the doodling stage, the creation of graphics can represent a serious stumbling block in putting together a computer game. Plenty of artists are willing to contribute their work for a fee, but if you're working on a tight budget that's only a small consolation. Fortunately, it's possible to construct striking images on a microcomputer even *without* notable artistic talent — if you've got the right software.

Some of that software is so expensive that you'd be better off hiring the artist. But not all graphics software costs the proverbial arm and a leg. Some of the best graphics creation software is free.

The three-dimensional maze images used in the BITMAZE program in chapter 4 were created entirely with freeware graphics utilities. I don't have room to include those utilities on the disk that comes with this book, but they can be obtained on most online services and quite a few local BBSes, as well as from computer user groups. And The Waite Group has published several book/disk combinations around these software packages. In the pages that follow, I'll describe how I created the images and show you how to create new images of your own.

POVRAY

The most exciting of these software packages in the Persistence of Vision Raytracer, usually called POVRAY for short. As you may recall from chapter 8, a ray tracer is a program that models scenes using mathematical descriptions of objects and light sources, and renders those scenes by tracing the path of imaginary rays of light from the viewer's eyes into the display. Most ray tracers can produce stunning images, and the images created by POVRAY can stand alongside the best of them. POVRAY was created by a team of programmers who can usually be found hanging around the Graphics Developers Forum (GRAPHDEV) on the CompuServe Information Service. Fortunately for the rest of us, these programmers aren't asking for any money from the users of POVRAY. The program is absolutely free.

All of the images in the BITMAZE program (with the exception of the on-screen compass) were rendered with POVRAY. To create an image with POVRAY, you first need to create an ASCII text file containing a description of the image using POVRAY's built-in descriptor language. The descriptor files used to create the BITMAZE images can be found in the BITMAP directory on the accompanying disk. There are ten of them, all with names ending in the file extension .POV. An example of one of these files, called TESTMAZE.POV, appears in Listing A-1.

Listing A-1 TESTMAZE.POV

```
// MAZE.POV

#include "colors.inc"
#include "textures.inc"
#include "stones.inc"
#include "maze.inc"

#declare MazeCube=
  object {
    box { <-1 0 -1><1 2 1> }
    texture {
      WallTexture
      scale WallTextureScale
      ambient AmbientLevel
    }
  }

// A camera pointed down an aisle of the maze
```

```
camera {
  location <0 1 -2>
  look_at<0 1 3>
}

object {MazeCube translate <-2 0 2>}
object {MazeCube translate <-2 0 4>}
object {MazeCube translate <-2 0 8>}
object {MazeCube translate <0 0 10>}
object {MazeCube translate <2 0 0>}
object {MazeCube translate <2 0 2>}
object {MazeCube translate <2 0 6>}
object {MazeCube translate <2 0 8>}

// The floor
object {
  plane {<0 1 0> 0}
  texture {
    FloorTexture
    ambient AmbientLevel
    phong 1
    scale FloorTextureScale
    reflection FloorReflection
  }
}

// The ceiling
object {
  plane {<0 1 0> 2}
  texture {
    CeilingTexture
    scale CeilingTextureScale
    ambient CeilingAmbientLevel
  }
}

object {
  light_source { <0 1 -2> color White}
}
```

It's not necessary to understand every line of this "program" to see what it does. Basically, it creates a series of texture-mapped blocks, distributes them at intervals on a texture-mapped reflective floor, throws some light on the scene, and renders it from the viewpoint of an imaginary camera. If you're curious to know exactly how this is done, you should locate a copy of the POVRAY documentation (or buy a copy of The Waite Group's *Ray Tracing Creations*) and read about the descriptor language.

For our purposes, the most important part of the descriptor file is at the beginning. You'll notice that the file begins by including several additional files, exactly as we've been doing at the beginning of our C++ programs. These files become part of the descriptor file, extending its capabilities. The first three

include files are standard POVRAY files and are distributed as part of the POVRAY package in the same way that include files like STDIO.H are distributed with every C/C++ compiler. The fourth include file, MAZE.INC, is one that I wrote. Actually, it's one of several files, all of which can bear the name MAZE.INC at some point in the rendering process. I'll explain that further in a moment. In the meantime, check out the sample MAZE.INC file in Listing A-2.

Listing A-2 MAZE.INC

```
#declare AmbientLevel=.45
#declare WallTexture=texture{Stone18 phong .1 turbulence 1}
#declare WallTextureScale=<.75 .75 .75>
#declare FloorTexture=texture{checker color Copper color White}
#declare FloorTextureScale=<.5 .5 .5>
#declare FloorReflection=.14
#declare CeilingTexture=texture{Bright_Blue_Sky}
#declare CeilingTextureScale=<1 1 1>
#declare CeilingAmbientLevel=1.0
```

As you can see, the MAZE.INC file is nothing more than a series of #declare statements. These work in POVRAY much as #declare statements work in C/C++ — that is, they declare a string of ASCII characters to be equivalent to a different string of characters. When the program is parsed (that is, read by the ray tracer), the first string is replaced by the second wherever it appears. This allows us to use meaningful terms such as WallTexture in place of such relatively incomprehensible expressions as texture(Stone18 phong .1 turbulence 1). And, if you'll glance back up at Listing A-1, you'll see that all of the terms defined in the MAZE.INC file are used in TESTMAZE.POV.

Why not just put the list of #declared terms in TESTMAZE.POV itself? Because keeping them in a separate file allows us to create new and very different looking mazes simply by creating new MAZE.INC files, or by editing old ones. There's no need to produce new versions of TESTMAZE.POV. If you'll look in the BITMAP directory, you'll see that there are several files with the extension .MAZ, such as GREEN.MAZ, ICE.MAZ, APOCALYP.MAZ, and so forth. These are MAZE.INC files in disguise. They can't be called MAZE.INC because there can only be one file of that name in the directory. But, by the simple expedient of typing COPY ICE.MAZ MAZE.INC, anyone of them can *become* the MAZE.INC file. I've included a batch file that will do just that, along with invoking POVRAY to render the maze described by that file.

Building a Maze

So how do you build your own set of maze graphics? First, get a copy of POVRAY, if you don't have one already. (The files in the BITMAP directory are intended for POVRAY 1.0, which has been superseded by POVRAY 2.0. If you want to use them with 2.0, you'll need to make some modifications to the batch files in this directory, as you'll see momentarily.) Once POVRAY is on your disk, you'll need to set it up properly for your system. That means you'll have to include the name of your POVRAY directory in the path statement in your AUTOEXEC.BAT file. And you need to configure POVRAY for your system by creating an environmental variable called POVRAYOPT. This variable will tell POVRAY, among other things, what kind of video adapter you're using. Instructions for doing this are in the POVRAY documentation. Be sure that the main POVRAY executable file is called POVRAY.EXE.

You'll also need to find a program called PICLAB. Like POVRAY, PICLAB is free and widely available from online services, BBSs, and user groups. You can also find it on the disk included with The Waite Group's book *Image Lab*, along with a stripped-down version of POVRAY. (Unfortunately, the version of POVRAY in *Image Lab* lacks one of the standard include files that you'll need.) Place PICLAB somewhere in your DOS path, making sure to give it the name PL.EXE.

Once POVRAY and PICLAB are in place, the rest is easy. Go to the BITMAP directory and type:

```
PREVIEW ICE
```

This executes a batch file called PREVIEW.BAT, which will render a single image of the maze described by the #defined constants in the file ICE.MAZ. To see how it does this, look at the text of PREVIEW.BAT in Listing A-3.

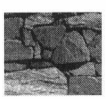
Listing A-3 PREVIEW.BAT

```
copy %1.maz maze.inc
povray +itestmaze.pov +otestmaze.tga -w188 -h120 -p
pl makemap testmaze
pl maketga testmaze
tgashow testmaze.tga
```

First, the batch file copies ICE.MAZ (or whatever .MAZ file you indicated when you typed PREVIEW) to MAZE.INC. Then it calls POVRAY to read the

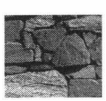
file TESTMAZE.POV (which you saw in Listing A-1) and creates a rendered image in Targa 24-bit format called TESTMAZE.TGA. This image will have a width of 188 pixels and a height of 120 pixels. The -p switch tells POVRAY not to pause after rendering the image. If you're using POVRAY 2.0 or later, you'll need to add the +MV1.0 switch immediately after the -p switch. This tells POVRAY 2.0 to accept a POVRAY1.0 file.

We can't use a 24-bit Targa file in a mode 13h program, so we'll have to convert it to an 8-bit Targa file. The batch file calls PICLAB to do this. For reasons that I'll explain later, this is done with two commands, PL MAKEMAP TESTMAZE and PL MAKETGA TESTMAZE. The first invokes PICLAB and tells it to read a file called MAKEMAP. This is a PICLAB batch file, which contains a set of commands for POVRAY to execute in its batch mode. The text of the MAKEMAP file appears in Listing A-4. The commands in the file tell PICLAB to create a 256-color palette for the TESTMAZE image and save it to the disk under the name MAZE.MAP. The next command, PL MAKETGA TESTMAZE, causes PICLAB to execute the commands in the MAKETGA batch file (shown in Listing A-5), which tells PICLAB to translate the TESTMAZE image to the 256-color palette created by the previous command and save it back to the disk under its original name, TESTMAZE.TGA.



Listing A-4 MAKEMAP

```
set fileformat "TARGA"
tload %1.tga
makepal
psave maze.map
```



Listing A-5 MAKETGA

```
set fileformat TARGA
load %1.tga
pload maze.map
map
save %1.tga
```

Finally, the batch file calls the TGASHOW utility that we created back in chapter 4 to display the 256-color image. Congratulations! You've now created a single view of a maze.

Of course, that's not the same thing as creating a set of maze graphics that you can animate with the BITMAZE program. But now that you've seen what the ICE.MAZ graphics look like using the PREVIEW batch file, you're ready to render the entire graphic set with the MAKEMAZE batch file. The MAKEMAZE batch file, which is in Listing A-6, does much the same sort of thing that the PREVIEW batch file does; it just does more of it. It can be invoked by typing MAKEMAZE ICE at the DOS prompt in the BITMAP directory.

Listing A-6 MAKEMAZE.BAT

```
copy %1.maz maze.inc
povray +ifront1.pov +ofront1.tga -w188 -h120 -p
povray +ifront2.pov +ofront2.tga -w188 -h120 -p
povray +ifront3.pov +ofront3.tga -w188 -h120 -p
povray +ifront4.pov +ofront4.tga -w188 -h120 -p
povray +ifront5.pov +ofront5.tga -w188 -h120 -p
povray +iside1.pov +oside1.tga -w188 -h120 -p
povray +iside2.pov +oside2.tga -w188 -h120 -p
povray +iside3.pov +oside3.tga -w188 -h120 -p
povray +iside4.pov +oside4.tga -w188 -h120 -p
pl makemap side1
pl maketga front1
pl maketga front2
pl maketga front3
pl maketga front4
pl maketga front5
pl maketga side1
pl maketga side2
pl maketga side3
pl maketga side4
```

Once again, you'll have to edit the batch file by adding the +MV1.0 switch after the -p switch in each of the nine lines that call POVRAY.

As before, the batch file copies ICE.MAZ (or whatever) to MAZE.INC and then calls POVRAY. However, this time it calls POVRAY nine times, to create the files FRONT1.TGA through FRONT5.TGA and SIDE1.TGA through SIDE4.TGA. You may recall from chapter 4 that these are the nine image files that the BITMAZE program slices and dices to create the animated image of the maze. The first five show the fronts of the maze "wall cubes" from five different distances and the last four show the sides of the cubes from four different distances. You can view these files using the TGASHOW utility. The descriptors for these images are in the files FRONT1.POV through FRONT5.POV and SIDE1.POV through SIDE2.POV. The text of FRONT1.POV is shown in Listing A-7. As you can see, it looks pretty much like TESTMAZE.POV; it even includes the same files, MAZE.INC in particular, at the beginning.

Listing A-7

```
// FRONT1.POV

#include "colors.inc"
#include "textures.inc"
#include "stones.inc"
#include "maze.inc"

#declare ZTRANS=2
#declare MazeCube=
  object {
    box { <-1 0 -1><1 2 1> }
    texture {
      WallTexture
      scale WallTextureScale
      ambient AmbientLevel
    }
  }

// A camera pointed down an aisle of the maze
camera {
  location <0 1 -2>
  look_at<0 1 3>
}

object {MazeCube translate <-2 0 ZTRANS>}
object {MazeCube translate <0 0 ZTRANS>}
object {MazeCube translate <2 0 ZTRANS>}

// The floor
object {
  plane {<0 1 0> 0}
  texture {
    FloorTexture
    ambient AmbientLevel
    phong 1
    scale FloorTextureScale
    reflection FloorReflection
  }
}

// The ceiling
object {
  plane {<0 1 0> 2}
  texture {
    CeilingTexture
    scale CeilingTextureScale
    ambient 1
  }
```

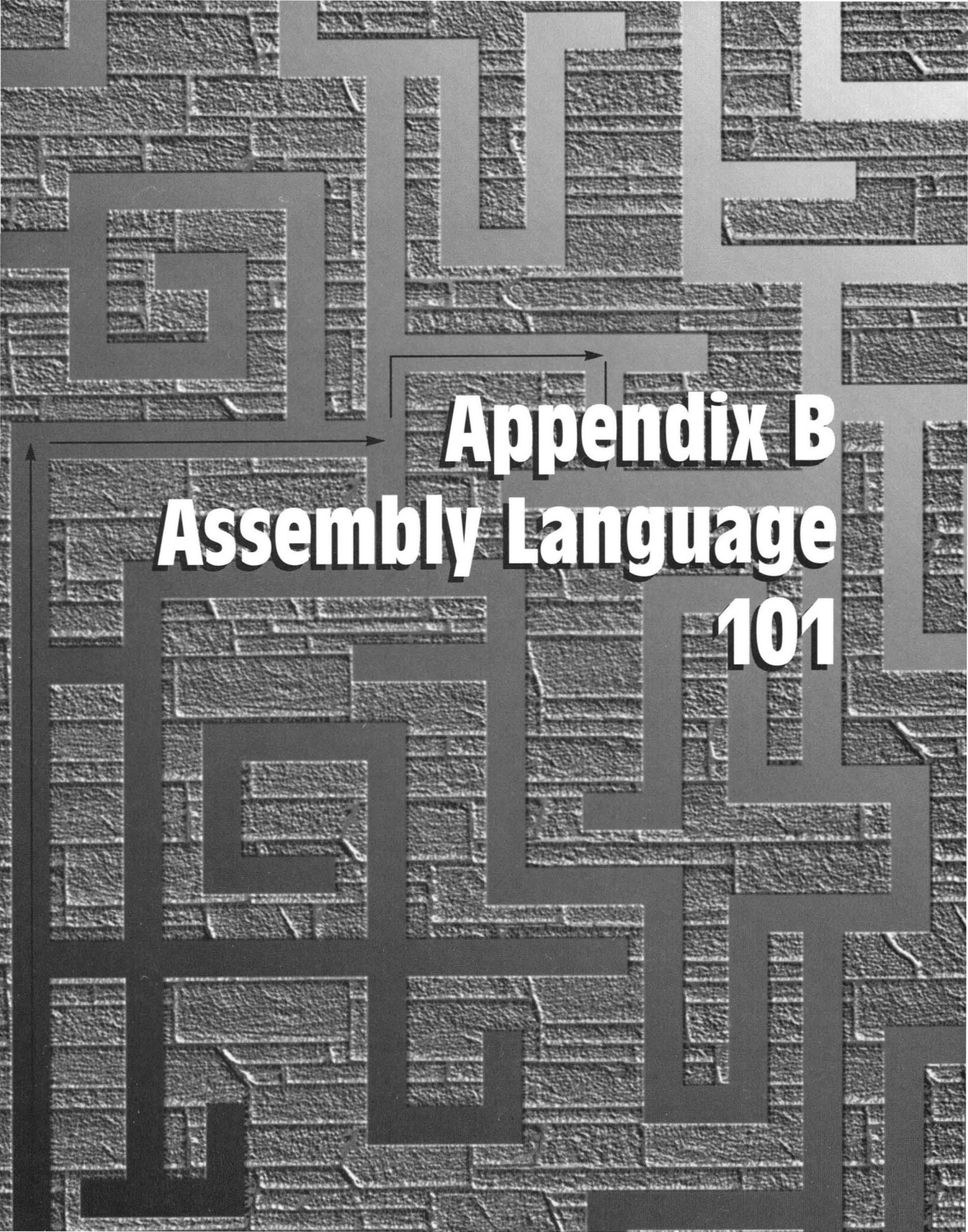
```
}

object {
  light_source { <0 1 -2> color White}
}
```

(Be warned that the process of creating a complete set of graphics files with POVRAY can be time consuming, especially if you're using an older machine. On my 40MHZ 80486 machine, the process takes about 15 minutes, but on older machines it will take longer. If you have a newer machine, on the other hand, the process will be faster.)

Once the 24-bit images are rendered, the batch file calls PICLAB to create a 256-color palette for them. It then calls PICLAB nine more times to convert all of the images to that palette. That's why the palette creation is a separate step. It allows all images to be mapped to the same palette, a necessity if we're going to be slicing and dicing them onto a single 256-color mode 13h display.

Now you can run BITMAZE and see what it looks like with the ICE.MAZ graphics. If you want to convert it back to its original graphics set, type MAKEMAZE GREEN. And if you want to explore the additional graphics sets I've included in the form of .MAZ files, just type MAKEMAZE <name>, where <name> is the name of the .MAZ file minus the .MAZ extension. Before you create a set of maze graphics, though, you might want to preview the .MAZ file by typing PREVIEW <name>, to see if you like it first.

Feel free to use these .MAZ files as templates for your own mazes. Load one of them into an editor, change some of the definitions, and save it to the disk under a new name (though you must remember to give it the .MAZ extension). Of course, this will require that you have at least a smidgeon of knowledge of the POV descriptor language, something I don't have space to go into here. But don't let that daunt you. Writing POV descriptors is simpler than you might think.



Appendix B Assembly Language 101

B

any otherwise competent programmers find assembly language baffling and intimidating. While it isn't necessary to master an assembler in order to write commercial game programs, a small knowledge of the subject would be useful in understanding some of the brief assembly language procedures in this book. This appendix won't teach you to write assembly language code, but hopefully it will give you a sufficient grasp of the subject to understand such code when you see it.

An assembly language program is written using two kinds of instructions: assembly language instructions and assembler directives. *Assembly language instructions* represent actual machine language codes that can be executed by the CPU after they are translated by a program called an assembler. *Directives* give the assembler important information about how it is to process the rest of the program.

Few programs are written entirely in assembly language. But it is quite common to write small assembly language modules to be included with a C++ program. These modules consist of a few assembly language procedures, or PROCs, which play roughly the same role as C++ functions. In fact, if properly written, these PROCs can be called from C++ exactly as though they were C++

functions. A PROC begins with the name of the procedure followed by the assembler directive PROC, like this:

```
AsmFunction PROC
```

And it must end with the name of the procedure followed by the assembler directive ENDP, like this:

```
AsmFunction ENDP
```

Between the PROC and ENDP directives you place a series of assembly language instructions, each of which occupies a single line of the assembly language source file. These instructions alternate with occasional assembler directives — the names of which are usually written in uppercase letters — and with comments, which must be preceded by a semicolon (;). (The semicolon in an assembler works exactly like the double slash (//) in C++, causing the assembler to ignore everything until the next carriage return.)

Each instruction consists of a *mnemonic*, which represents a CPU operation, followed optionally by one or more operands indicating what memory locations are to be operated upon. The mnemonic is usually a three-to-six-letter word, such as MOV or PUSH or ADD. Here's a typical assembly language instruction, followed by a comment:

```
add ax,[length]  ; Add AX and LENGTH, leaving sum in AX
```

Most of the assembly language instructions in the PROC will concern themselves with moving numeric values around in the computer's memory and performing mathematical and logical operations on those values. Often, these instructions reference a special set of memory locations situated in the computer CPU itself. These memory locations, known as the CPU *registers,* can be considered a set of permanent integer variables. The registers we will be concerned with are named AX, BX, CX, DX, SI, DI, BP, SP, ES, DS, CS, and SS. Four of these registers — AX, BX, CX, and DX — can be broken in two and treated as pairs of 8-bit variables. AX becomes AH and AL (with AH the high-byte of AX and AL the low-byte), BX becomes BH and BL, CX becomes CH and CL, and DX becomes DH and DL. The CPU registers are diagrammed in Figure B-1.

Assembly Language Pointers

Most of the CPU registers can be used for performing general arithmetic operations, such as addition and subtraction, and some of the 16-bit registers can be used as pointers to locations elsewhere in the computer's memory. Just as

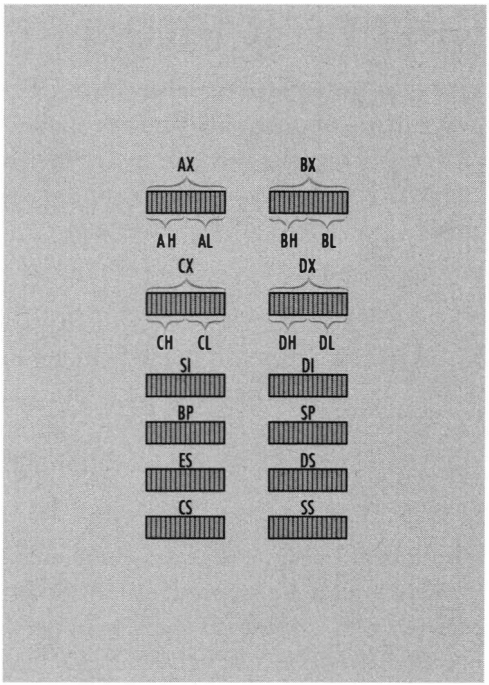


Figure B-1 Major CPU registers

preceding the name of a pointer variable in C++ with the indirection operator (*) indicates that we are referencing the memory location to which that pointer is pointing, so surrounding the name of a CPU register with square brackets ([ ]) indicates that we are referencing the memory location to which that register is pointing, like this:

```
[bx]
```

Actually, we will need to use *two* CPU registers to point to a memory location, with one register holding the segment portion of the address and another holding the offset portion. The segment portion must always be placed in a *segment register,* usually either DS or ES. You include the name of this register in brackets with the name of the register holding the offset, like this:

```
[es:dx]
```

When the segment is in DS, however, its name does not have to be explicitly mentioned in brackets, since the CPU uses the segment in DS by default for all data references.

Getting It into a Register

How do you place a value in a CPU register? In C++, you would use the assignment operator (=), but this does not exist in assembly language. Instead, you must copy the value from another memory location or CPU register by using the MOV instruction (which should really be called the COPY instruction), like this:

```
mov ax,[es:dx]
```

This copies to the AX register the value stored in the memory location pointed to by [es:ds]. We can also identify memory locations by their addresses, though it is more common to name those addresses and then identify the locations by name. For the moment, I won't discuss techniques for assigning names to locations, but you would refer to a named location like this:

```
mov cx,size
```

where *size* is a name that we've assigned to a 16-bit memory location.

Values can be copied not only from memory locations, but from other registers and even from the instruction itself. A value copied from an instruction is called an *immediate value* and is referenced like this:

```
mov dx,1040h
```

This copies the value 1040h (the "h" indicates that the value is in hexadecimal) from the instruction into the DX register.

When using assembly language pointers, it is sometimes necessary to specify what sort of value the pointer is pointing at. The three types of values recognized by 80x86-series CPUs are BYTEs (8-bit values), WORDs (16-bit values), and DWORDs (Double WORDs, or 32-bit values). Thus, a pointer is either a BYTE PTR, which points at a BYTE value; a WORD PTR, which points at a WORD value; or a DWORD PTR, which points at a DWORD value. For instance, these instructions copy a WORD value from the address pointed to by es:bx to the AX register:

```
mov ax,word ptr [es:bx]
```

Of course, the assembler can also figure out that [es:bx] must be a word pointer because you want to move the value stored there into a 16-bit register. But there will be instances when the size of the value is less obvious. And it doesn't hurt to make it explicit anyway, if only to make your code clear to somebody else reading it.

There are special assembly language instructions for putting pointers into registers. An assembly language pointer actually stretches across two registers,

with either ES or DS holding the segment part of the pointer and another 16-bit register holding the offset. Does this mean that we must use two MOV instructions to get the pointer into these two registers? Fortunately not. The 80x86 instruction set (that is, the set of instructions that can be executed by CPUs in the 80x86 series) includes special instructions for loading pointers into a pair of registers. The two instructions that will be used in the programs in this book are LDS and LES. The first loads a segment into the DS register and an offset into a second register. The other does the same with the ES register. For instance, if we wished to load the pointer stored at the address we've named *old_pointer* into the es:dx registers, we could write

```
les ds,old_pointer
```

Assembly Language Odds and Ends

Before you can use certain registers in a PROC that is to interface with C++ code, you must save the value stored in those registers in a section of memory called the stack. This is done by using the PUSH instruction, like this:

```
push ds
push di
push si
```

Then, before the program ends, these values must be restored to the original registers, in reverse order, by using the POP instruction, like this:

```
pop si
pop di
pop ds
```

Some assembly language instructions have an effect on the CPU *flags*. These are simply isolated bits within a special register, which reflect the results of instructions. For instance, if the result of a subtraction instruction (SUB) is zero, the *zero flag* will be set (given a value of one). What good does this do you? You can give other 80x86 instructions contingent on the way in which the flags are set. These instructions usually cause the CPU to begin executing instructions at a different address in memory if the flags have a particular setting. Thus, they can be used to simulate high-level IF and WHILE instructions, by skipping some instructions and executing others based on the outcome of other operations or repeating a set of instructions in a loop if the flags are set in a certain way. Commonly, these operations are preceded by a comparison instruction (CMP), which checks to see if two values are equal or are unequal in a certain way. For instance, the instruction *cmp ax,dx* would compare the values in the AX and DX

registers. This could then be followed by a *jz* instruction, which would cause the processor to "jump" to a new address if the zero flag is set — that is, if the values in AX and DX are equal. This is equivalent to the C++ instruction: *if (ax==dx)*.

Perhaps the fanciest of these "conditional jump" instructions is LOOP, which subtracts one from (*decrements*) the value in the CX register and jumps to a certain address if CX doesn't then equal zero. Variations on this instruction, such as LOOPNE and LOOPZ, can be made to check the value of a flag and terminate the loop when a certain flag setting is detected, even if CX has not yet been decremented all the way to zero. These instructions can be used to produce structures similar to the high-level WHILE and FOR loops.

Occasionally, assembly language procedures must call other procedures, the way that C++ functions call other functions. This is done via the CALL instruction, which must be followed by the name of the procedure being called, like this:

```
call other_procedure
```

The assembler translates the name of the procedure being called into the actual address at which the code for that procedure resides in memory.

Finally, every assembly language procedure must end with a RET instruction and the ENDP directive. Although ENDP tells the assembler that the procedure is over, the RET instruction causes the CPU to return control of the program to C++ when the assembly language procedure is complete. The RET instruction is equivalent to the C++ return instruction, as well as to the final curly bracket (}) at the end of a C++ function.

There is a great deal more to know about assembly language, and we can only scratch the surface in this book. To avoid confusion, I'll explain the few relevant assembly language concepts as they arise. For now, we will mention only one more: how to pass parameters from C++ to an assembler.

Passing Parameters to Assembly Language Procedures

In C++, you can pass parameters to a function by placing the values of those parameters in parentheses after the function call, like this:

```
get_lost(23, "skidoo");
```

You'll pass parameters to assembly language procedures in exactly the same way. But how does the assembly language function *receive* the parameters? In C++, you would simply write a parameter list in the function header, specifying

the types of the parameters and the names by which you wished to refer to them. In an assembler, it's not quite that easy. When the assembler procedure begins, the parameters will be in the stack. To fetch the parameters from the stack, you need to know exactly how C++ organizes those parameters on the stack. But Turbo Assembler (and most other Microsoft-compatible assemblers for IBM-compatible computers) provides a simple method of accessing parameters: the ARG directive.

No, ARG isn't what you say when you stub your toe against your computer desk. It's a directive that tells the assembler which parameters you expect to be passed from C++, the order in which those parameters will be passed, and the number of bytes that each parameter will occupy. In return, the assembler allows you to refer to those parameters by name rather than by their location on the stack. The ARG directive works like this:

```
ARG first_param:BYTE, second_param:WORD, third_param:DWORD;
```

This tells the assembler to expect three parameters — *first_param, second_param,* and *third_param* — to be passed to the procedure from C++. It also tells the assembler how much memory each of these parameters will occupy. The first will occupy one byte (as indicated by the word BYTE attached to the name of the parameter by a colon), the second two bytes (or one WORD), and the third four bytes (or one DWORD). Other sizes can be specified, but these are the only ones that we'll use in this book.

Once the ARG directive has been placed at the beginning of the procedure, just after the PROC directive, you can refer to these parameters by name. The one catch is that the following pair of instructions must be included at the beginning of any procedure that uses the ARG directive:

```
push bp
mov  bp,sp
```

Place these two instructions before any other PUSH instructions are executed. These instructions set up the BP register as a pointer to the stack area where the parameters are stored. The assembler will translate all references to these parameters into pointers using the BP register. For this reason, you should never use the BP register in any procedure that has an ARG directive.

Near the end of the procedure, after all other POP instructions have been executed, include the following instruction:

```
pop bp
```

This restores the BP register back to the value it was previously storing. (Most likely, the C++ function that called the assembly language procedure was *also* using the BP register as a pointer to parameters stored on the stack, which is why it is important that the value be returned intact at the end of the procedure.)

Is it possible for an assembly language procedure to return a value the way that a C++ function can? Sure it is. An assembly language procedure can do anything that a C++ function can (and occasionally a bit more). The way in which a value is returned from an assembly language procedure depends on what type of value it is. For instance, values of type *int* (or any other 16-bit values) are returned in the AX register: The value must be placed in the AX register before the procedure terminates. Larger and smaller values are returned using similar techniques.

Index

#declare statements, 556
#defined constants, 557
#include statements, 554-555
* (dereferencing operator), 20
* (indirection operator), 565
// (double slash) comments, 26, 564
; (semicolon) comments, 24, 564
<< operator, 471
>> operator, 115, 471
16:16 numbers, 469
360-degree system, 248

A

absolute angle, 309
absorbed light, 262
addresses, 18-19
ambient light, 269, 419, 517
ANDing, 301, 304-305, 328
animation, 14, 240, 325, 461, 466
animation engine, 9-10
animation, and heightmapping, 397-398
animation, and lightmapping, 458
animation, maze, 97-103
ARG directive, 22, 81
argument parameters, 117-118, 322
assembly language, 17, 22, 563-570
assembly language translation, 461-462, 519
automap, 546-548
automap( ) function, 547

B

background color, 269
background images, 218
banding, 285, 431
Bard's Tale games, 5
bg (background) PCX structure, 150
binary numbering, 468, 471
bitmap defined, 108
bitmap scaling, 196-206
bitmap storage, 108-109
bitmapped drawmaze( ) function, 140-149
bitmapped maze, building, 129-157
bitmapped mazes, 12-13
BITMAZE program, 149-157
BITSCALE.CPP, 202-206
blit( ) function, 151-152, 158
block-aligned heightmaps, 360-371
block-aligned lightmaps, 433
BLOKCAST.CPP, 363, 371-377
BLOKDEMO.CPP, 378-381
Borland C++, 17
Borland Graphics Interface (BGI), 17, 32
botdiff, 176
boterror, 177
bottom[ ] array, 186
bounds checking, 346, 364
break code, 70-72
break instruction, 53
Bresenham's Algorithm, 40, 43-46, 173, 177, 299
brightness, 401-404, 406-407, 419
bubble sort, 535-536

C

calculations, math, 464-465
Cartesian coordinate system, 32-33, 241, 250, 299

Cartesian plane, 232-234
Cartesian units (units of measure), 241
ceil() function, 285
ceiling[ ] array, 338-339, 344, 347
ceilingcasting, 338, 343-344
ceilinglites[ ] array, 433-434, 458
ceilings, optimized, 489-497
circular system, 248
clipping wall, 369
CLK_TCK macro, 98
clock() function, 98
clrscr() function, 83
CLS (clear screen) procedure, 31-32
CMP instruction, 71
color (palette), 27-30, 268-269
color brightness, 407
color and light, 262-264
color map, 111-112, 114-115
column angle, 308
column clipping, 328-330
COLUMNLOOP macro, 487, 489
comments (; and //), 24, 26, 564
compass, 248
compass PCX structure, 150
compass_face array, 150, 153
components, vector, 247
compressing data, 109, 121, 159
computer role-playing games (CRPG), 4
continuous (smooth) motion, 7, 325
coordinate pair, 32
coordinate systems
    Cartesian, 32-33, 241, 250, 299
    fine, 300-301, 311
    polar, 247-258
cosine adjustment, 315-316, 342
cosine/sine, 254-256, 476-478
CPU flags, 567
CPU registers, 564-567
credits, opening, 541-546

D

decimal numbering, 468, 470
delete function, 153
depth sort, 535-536
dereferencing, 20
direction vectors, 241-247, 250-252
discriminant, 278, 283
display_slice( ) function, 146
distance[ ] array, 527-528
dithering, 285
divide-by-0 error, 273, 284, 304, 312, 405
division (/) operator, 29
Doom, 9-10, 359-360, 396-397
draw_maze( ) function. *See also* drawmaze( ) function
    block-aligned heightmapped, 371-377
    floorcasting, 344-346
    floorcasting, wallcasting, 348-354
    lightmapped, 434-440
    lightsourced, 423-430
    optimized, 500-508
    texture-mapped ray-cast, 332-336
    tiled heightmapped, 389-392
    tiled lightmapping, 446-453
    wallcasting, 306-321
draw_slice( ) function, 149
drawbg( ) function, 218
drawbox( ) function, 50, 184
drawceilrow( ) function, 491, 499
drawfloorrow( ) function, 491, 498
drawmaze( ) function, 99. *See also* draw_maze( ) function
    bitmapped maze, 140-149
    polygon maze, 185-190
    wireframe 3D maze, 48-58
drawobject( ) function, 537-539
drawplayer( ) function, 547
drawwall( ) function, 484-488
Dungeon Master, 6, 76
DW (Define Word) directive, 67

E

EMPTY.TGA file, 218
endpoints, 36-37
eraseplayer( ) function, 548
error code, 81-82, 113, 117-118, 124
error message, 414
error term, 43, 178-179, 330-332
error terms, 173, 177, 199-201, 210-212
escape clause, 74

event manager, 91-97
event structure, 92-93, 98
events, maze, 92-93
exit( ) function, 118
expanded memory (EMS), 159-160
Eye of the Beholder, 7, 107

F

fabs( ) function, 413
field of view, 235
file formats, 109-110
fine coordinate system, 300-301, 311
fish-eye effect, 315, 342
fixdiv( ) function, 475-476
fixed point math, 174, 312, 465-476, 519
fixed point trigonometry, 476-481
fixmul( ) function, 473-474
flicker, 100-101
flight simulators, 7-8, 231, 234, 239-240
floating point math, 173-174
floating point unit (fpu), 465-466
floating point variables, 311
floor( ) function, 285, 339, 342
floorbase[ ] array, 384-385
floorcasting, 298, 337-346
FLOORDEM, 346
floorheight changes, 365-371
floorheight[ ] array, 361-364, 367
floorheight[ ] array, tiled, 384
floorlites[ ] array, 433-434, 458
FLOORLOOP macro, 493, 499
floors, optimized, 489-497
floor[ ] array, 342, 347
flor[ ] array, 338-339, 342, 362-363
flor[ ] array, tiled, 385-386
for( ) loop, 27
for( ) loops, nested, 29, 118. *See also* loops, nested
fractions, 468
freeware, 553, 557
function 25h, 67
function 35h, 67

G

game features, 539-541
gameport byte, 85-86
Gamers Forum, xiii, xviii
GARDENS.CPP, 549
Get Interrupt Vector, 67
getevent( ) function, 95-97
GIF format, 109, 110-111
GO GAMERS, xiii, xviii
Gold Box games, 5
GOMAZE program, 101-103
gotoxy( ) function, 83
grab( ) function, 150-151, 158, 197
GRAPHDEV forum, 554
graphics utilities, 553
gray-scale palette, 268
grid lines, 144-145, 302

H

hardware, 462-466
header files, 26
height changes, floor, 365-371
height tiles, 383-384
height variable, 177
height of viewer, 364-365
heightcasting, 363-364, 386, 395-396
heightmapped ceilings, 381
heightmapping, 9, 354, 360
heightmapping, animated, 397-398
heightmapping engine, 386-389
heightmapping, hybrid, 396-397
heightmaps, block-aligned, 360-371
heightmaps, pixel-aligned, 382
heightmaps, tiled, 382-389
hidden surface removal, 324, 382, 524-528, 535, 539
HIGHMAPS.PCX, 383
HORZLINE program, 38-39
HTIMER routines, 508-510

I

if( ) statement, 179, 200
increment value, 485
incremental division, 43, 196, 330
init_events( ) function, 93
initkey( ) function, 67-69

initmouse( ) function, 78-79, 81-82
inline function, 479
input devices, 63
insertion sort, 535-536
installation, xix
INT 21H routines, 67
INT 33H instruction, 77
interface types, 541
interrupt handler, 65-67, 69
interrupt, keyboard, 65
inverse square law, 404-405
IRET instruction, 69, 73

J

joystick, 84-91
joystick buttons, 85-86
joystick calibration, 89, 93-95
joystick position, 86-88
JOYSTICK program, 89-91
jump instruction, 71-72, 568

K

kbhit( ) function, 27, 542
keyboard, 63-76
keybuffer array, 93, 96
kilobyte, defined, 230
Kirk, James T., 231
kludge, 523-524

L

libraries of functions, 17, 64
light, 402-404
light and color, 262-264
light intensity, 405-409, 419, 432
light pool, 403, 409, 446
lightmapping, 432-434
lightmaps, block-aligned, 433
lightmaps, tiled, 445-446
lightmaps, using, 458
lightsource( ) function, 279-280, 282, 284-285
lightsource[ ][ ] array, 409, 414, 423
lightsourced 3D graphics, 9
lightsourcing, 190-191, 226, 279, 401-404
lightsourcing, alternate effects, 430-432
lightsourcing formula, 405-414, 418
lightsourcing tables, 409-410, 417-419
line drawing, 35-43
line, horizontal, 37-39
line, vertical, 39-40
linedraw( ) function, 44-46
lines, 37-43
lines of arbitrary slope, 40-43
listings
    bitmapped drawmaze( ) function, 147-149
    BITSCALE.CPP program, 202-206
    blit( ) function, 152
    block-aligned heightmapped draw_maze( ) function, 371-377
    BLOKDEMO.CPP, 378-381
    CLS (clear screen) procedure, 31-32
    draw_slice( ) function, 149
    drawfloorrow( ) function, 498
    drawwall( ) function, 488
    fixdiv( ) function, 476
    fixmul( ) function, 474
    floorcasting draw_maze( ) function, 344-346
    floorcasting, wallcasting draw_maze( ) function, 348-354
    FRONT1.POV, 560
    getevent( ) function, 96-97
    grab( ) function, 151
    HORZLINE program, 38-39
    init_events( ) function, 93
    initkey( ) function, 68-69
    initmouse( ) function, 79
    JOYSTICK.CPP program, 90-91
    lightmapped draw_maze( ) function, 434-440
    lightsourced draw_maze( ) function, 423-430
    linedraw( ) function, 44-46
    LITEDEMO.CPP program, 419-422
    load_image( ) function, 126-127
    load_palette( ) function, 127
    loadPCX( ) function, 125
    loadTGA( ) function, 115-117
    main( ) function from BITMAZE.CPP, 154-157

listings *(continued)*
main( ) function from GOMAZE.CPP, 101-103
MAKELITE.CPP program, 415-417
MAKEMAP, 558
MAKEMAZE.BAT, 559
MAKETGA, 558
MAKETRIG.CPP, 481-483
MAPDEMO.CPP program, 441-445
MAZE.CPP program, 53-58
MAZE.INC, 556
MOUSE.CPP program, 83-84
newkey( ) function, 73
OPTDEMO.CPP module, 510-517
optimized draw_maze( ) function, 501-508
PALETTE.CPP program, 29-30
PCX_HEADER structure, 121
PCX_STRUCT structure, 122
PCXSHOW program, 128-129
POLYCLIP program, 184-185
POLYDEMO.CPP program, 183-184
polydraw( ) function, 180-182
POLYGON.CPP, 165-166
polygon-based drawmaze( ) function, 188-190
polytext( ) function, 212-215
portions of TRIG.CPP, 484
PREVIEW.BAT, 557
readjbutton( ) function, 86
readmbutton( ) function, 79-80
readstick( ) function, 87-88
relpos( ) function, 81
remkey( ) function, 69
RTDEMO.CPP program, 285-290
SCANKEY.CPP program, 75
setcenter( ) function, 94
setmax( ) function, 94-95
setmin( ) function, 94
SETMODE procedure, 24
SETPALETTE procedure, 28
TESTMAZE.POV, 554-555
TEXTDEMO.CPP program, 215-217
TEXTMAZE.CPP program, 218-226
texture-mapped ray-cast draw_maze( ) function, 332-336
tga_header structure, 111
TGASHOW.CPP program, 119-120
tiled heightmapped draw_maze( ) function, 389-392
tiled lightmapping draw_maze( ) function, 447-453
TILEDEMO.CPP program, 392-396
TLMDEMO.CPP program, 453-457
TRIG.H file, 483-484
VERTLINE program, 39-40
wallcasting draw_maze( ) function, 318-321
WALLDEMO.CPP program, 323-324
WHITEOUT.CPP program, 26
LITEDEMO.CPP, 419-422, 430
litelevel[ ] array, 417- 418, 423
LITEMAP.CPP, 434-440
load_image( ) function, 124-127
load_palette( ) function, 124, 127-128
loading Targa files, 112-117
loadPCX( ) function, 124-125, 153, 326
loadTGA( ) function, 112-117, 153
loop unrolling, 463-464, 519
LOOPNE instruction, 87
loops, nested, 29, 118, 365, 411, 466-467
low-resolution ray tracing, 268
lseek( ) function, 113

M

macro CLK_TCK, 98
macro COLUMNLOOP, 487, 489
macro FLOORLOOP, 493, 499
macro MK_FP, 20, 27
macros, 464, 478-479, 487, 519
magnitude, 241, 243-248
main( ) function, 26
make code, 70-72
MAKELITE.CPP, 410, 414-417, 431
MAKETRIG.CPP, 481-483
MAPDEMO.CPP, 441-445
math, 173-174
matrix[ ] array, 185-186
maze animation, 97-103
maze elements, 131
maze engine, refining, 158-160
maze games, 4
maze grid, 144-145, 298, 300-301

maze touring, 157-158, 160
maze types, 12-14
maze views, 131-133
MAZE.CPP, 53-58
maze-drawing algorithm, 134-140
megabyte, defined, 229-230
memory circuit, 18
memory problems, 158-160
messages to user, 410-414
mickey (unit of measure), 78, 80
MK_FP macro , 20, 27
mode 13h, 21
modulus operation, 29, 301
MONA.TGA, 196
mouse, 76-84
mouse driver, 77-79
MOUSE program, 82-84

N

new function, 114
newkey( ) function, 67, 69-73
normal vector, 244, 266
normalizing, 252
NOT instruction, 85
number line, 230
numbering systems, 18-19, 468

O

objdraw( ) function, 528
object array, 524, 528
objects, drawing, 531-534
objects, maze, 523-524
objects, representing, 528-531
objlist array, 534, 537
offset, array, 27
offsets, 18-19
open( ) function, 113
operator precedence, 246
OPTDEMO program, 508-517
optic parallax, 234
optimization review, 517-520
optimize, where to, 466-467, 520
origin, 35

P

paint programs, 108, 121, 163, 383
Painter's Algorithm, 535
palette, 27
PALETTE.CPP program, 29-30
palfadeout( ) function, 542
parameterized form, 277
PCX format, 109, 121-129
PCX_HEADER structure, 121
PCX_STRUCT structure, 122
PCXSHOW program, 128-129
perror( ) function, 414
perspective, 234-238, 316, 368
perspective equation, 533
Piclab, 110, 121, 198, 557
pixel, 21, 35-36
pixel, drawing, 201-202
pixel, plotting, 33-35
plane, defined, 229
player, tracking, 548-549
plot( ) function, 276, 285
point-slope equation, 174-175, 302-304, 313-314
pointers, 19-20
points, 35-36
polar coordinate system, 247-258
POLYCLIP program, 184-185
POLYDEMO program, 182-184
polydraw( ) function, 171, 180-182, 186
polygon clipping, 169-171, 173-174, 176
polygon, filled, 166-168
polygon graphics, 164
polygon maze, building, 171-182
polygon mazes, 12-13
POLYGON.CPP program, 165-166
polygon-drawing algorithm, 171-173
polygon-fill graphics, 167, 238
polygons, 164-168
POLYMAZE.CPP program, 190-191
POLYRAY ray tracer, 291
polytext( ) function, 208-215
polytype structure, 169, 171
ports, inputting data, 70, 85
POVRAY ray tracer, 108, 110, 121, 130-131, 291, 554-556

prev_t variable, 280, 282
PROC (procedure) statement, 22-24
processors, 462, 464-466, 470
profiler, 467, 520
ptr variable, 178
putwindow( ) function, 101
Pythagorean method, 245, 251, 275, 284, 314-315, 533

Q

quadratic equation, 278, 283

R

radian system, 249
radiation (light), 402-404
random number generator, 272
random( ) function, 285
randomize( ), 272
ratio variable, 209-210
ray casting, 292, 297-298, 303-305, 311-313
ray tracer, building, 264-280
ray tracer, defined, 131
ray tracers, 291
ray tracing explained, 262-264
ray tracing a maze, 130-131
ray-cast graphics, 258
ray-cast mazes, 12-14, 324-325
ray-casting algorithm, 298, 303-305
ray-object comparisons, 276, 280, 295-296
ray-wall comparisons, 304-305, 309
RAYDEMO, 354
rays, reducing, 296-297
read( ) function, 111, 113-114
readjbutton( ) function, 86, 88
readmbutton( ) function, 78-80, 82-83
readstick( ) function, 87-89
realism, 261
recursion, 134, 139, 141
recursive algorithm, 186
recursive function, 134, 140
reflected light, 262, 403
reflective surfaces, 291
register, 23
relpos( ) function, 78, 80-83
remkey( ) function, 67, 69, 76
ROM BIOS, 22-23
roots (quadratic), 278, 283
rotating bitmaps, 196
rotating point, 254-257
rotating ray, 309-311
rotation equation, 310-311
rotation, three-dimensional, 257-258
RTDEMO.CPP program, 285-290
run-length encoding, 122-123

S

scaling bitmaps, 196-206
scan code, 65, 70-72
SCANKEY program, 74-75
SCREEN.ASM, 22
screen-clearing, 30-32
segments, 19
semicolons, use of, 24, 564
Set Interrupt Vector, 67
set_palette( ) function, 118
setcenter( ) function, 93-94
setmax( ) function, 93-95
setmin( ) function, 93-94
SETMODE procedure, 24
setmode( ) function, 27
SETPALETTE procedure, 28
shadows, 269, 403, 419, 430
shift operation, 301, 311, 471-473
similar triangles method, 340-343, 364, 388
sine/cosine, 254-256, 476-478
sizeof( ) function, 113
slice-and-dice maze generation, 134, 163-164
slope, 40-43, 174-175
slope of ray, 312
smooth movement, 7, 325
smooth-scrolling arcade games, 10
sort types, 535-536
sphere, ray tracing, 282-284
sqrt function, 246
stack, 524, 567
string instructions, 31
surface normal, 266
swapping method, 535-536
switch( ) statement, 51, 281

T

t unit (unit of measure), 277, 283
table look-ups, 464-465, 476-478, 519
target hardware, 462
TEST instruction, 87
TEXTCAST.CPP, 332-336
TEXTDEMO, 336
TEXTDEMO.CPP, 215-217
textdemo( ) function, 215
TEXTMAZE.CPP, 218-226
texture clipping, 209-211
texture mapping, 164, 191, 195, 206-208, 291, 326-336
texture-mapped maze, 218-226
texture-mapped polygon function, 208-215
TGA (Targa) format, 110-120
tga_header structure, 111
TGASHOW program, 117-120
three-dimensional graphics, 226
three-dimensional space, 231-234
tick (unit of measure), 98
tiled heightmapping engine, 386-389
tiled heightmaps, 382-389
tiled lightmaps, 445-446
TILEDEMO.CPP, 392-396
TLMDEMO.CPP, 453-457
tmcolumn variable, 328
topdiff, 176, 179
toperror, 177, 179
top[ ] array, 186
trace_ray( ) function, 276-285
transformations, coordinate, 255-258
transforming, 256
translating point, 256
triangles, similar, 340-343, 364, 388
TRIG.CPP, 484
TRIG.H file, 483-484
trigonometry, 308-309, 465, 476-481, 519
Turbo Assembler (TASM), 22
Turbo Profiler (TPROF), 467, 520
two-and-a-half dimensions, 238-241
type override, 271

U

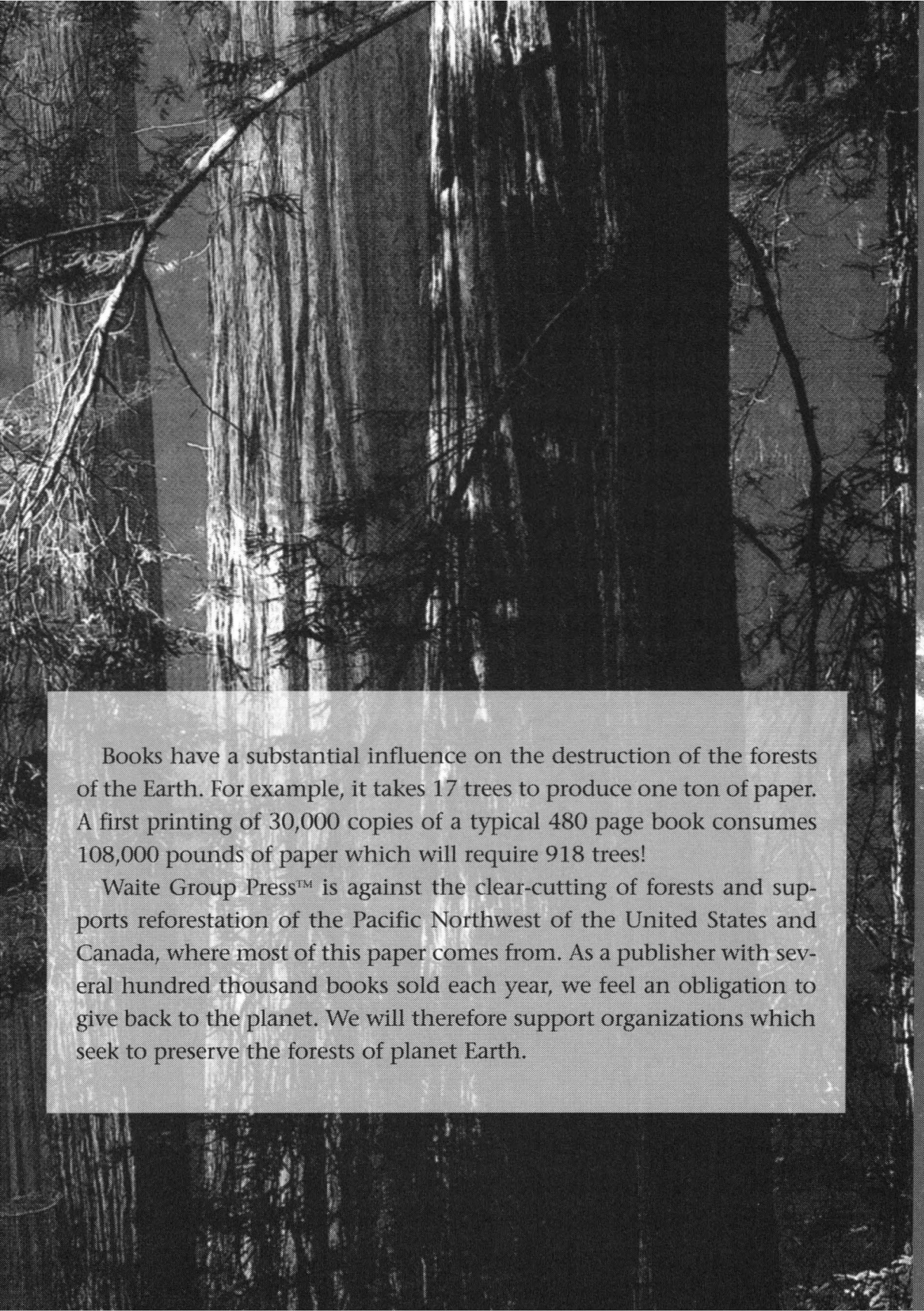
Ultima games, 4-5
underscore, use of, 22, 474
union structure, 265, 271
unit circle, 253-254
unit vector, 252-253

V

vector components, 247
vector_type structure, 265
vector_type variables, 272
vectors, direction, 241-247
vectors, unit, 252-253
vertex/vertices, 164, 238
VERTLINE program, 39-40
VGA palette, 27-30
video display programming, 17
video memory, 18, 24-27
video mode setting, 22-24
viewer height, 364-365
viewer orientation, 48-49
viewer position, 48-49
viewport, 169
virtual reality, 7-8, 523
visibility array, 524, 548-549
visibility variable, 51
visible[ ][ ] array, 527
VIVID ray tracer, 291

W

wall squares, 46-47
wallcasting, 298-300, 306-321
WALLDEMO.CPP program, 321-325
walls, optimized, 484-488
wall[ ] array, 347, 362-363
wall[ ] array, tiled, 385-386
while( ) loop, 153-154
while( ) statement, 179, 200
WHITEOUT.CPP program, 26
width variable, 176
wireframe mazes, 4, 12, 32-59
Wizardry games, 5, 107
Wolfenstein 3D, 9-11, 337-338

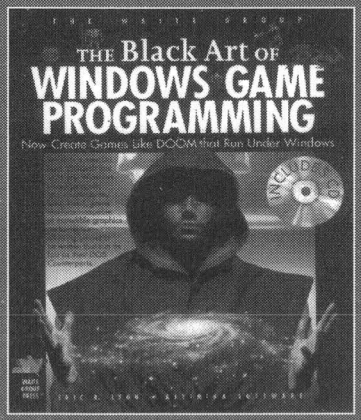


ENVIRONMENTAL AWARENESS

Books have a substantial influence on the destruction of the forests of the Earth. For example, it takes 17 trees to produce one ton of paper. A first printing of 30,000 copies of a typical 480 page book consumes 108,000 pounds of paper which will require 918 trees!

Waite Group Press™ is against the clear-cutting of forests and supports reforestation of the Pacific Northwest of the United States and Canada, where most of this paper comes from. As a publisher with several hundred thousand books sold each year, we feel an obligation to give back to the planet. We will therefore support organizations which seek to preserve the forests of planet Earth.

WAITE GROUP PRESS™

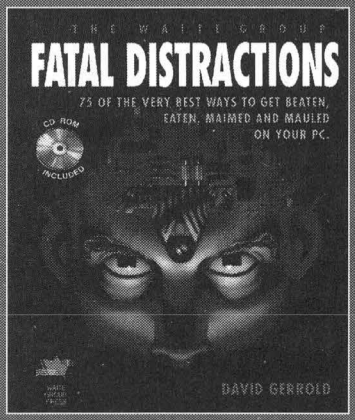


THE BLACK ART OF WINDOWS GAME PROGRAMMING

Eric R. Lyons

Microsoft's recently created toolkit, WinG (for Windows Games), is aimed specifically at making Windows games perform as fast as DOS-based games—without sacrificing any of the benefits of Windows. With WinG and the *Black Art of Windows Game Programming* you're in game programmers' heaven—your games will scream. That's what this book is about—quickly getting up to speed on the Windows programming environment and, specifically, taking advantage of WinG to make high-performance games.

Available February 1995
300 pages
ISBN: 1-878739-95-6
US $34.95 Can. $48.95

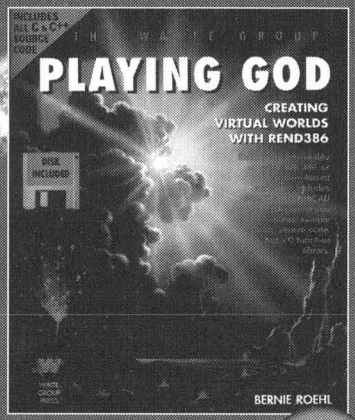

THE WAITE GROUP
FATAL DISTRACTIONS
75 OF THE VERY BEST WAYS TO GET BEATEN, EATEN, MAIMED AND MAULED ON YOUR PC.
CD ROM INCLUDED
DAVID GERROLD

FATAL DISTRACTIONS

David Gerrold

These top PC shareware games on CD-ROM will give you the thrills and chills of an arcade in your home without the embarassment of having total strangers overhear your agonized cries of defeat and your ecstatic screams of victory. These games include classics and little known adventure, simulation, combat, arcade, and dungeon games, as well as (for the adrenaline-challenged) table, word, card, and maze adventures.

Available now • 150 pages
ISBN: 1-878739-77-8
U.S. $26.95 Can. $37.95
Disk: 1–CD-ROM



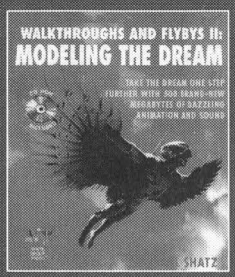
PLAYING GOD

BEST SELLER

Bernie Roehl

Who says it takes six days to create a world? With the guidance and software included with *Playing God*, you can make virtual worlds in a few hours, then walk, run, or fly through them. The book covers such topics as color systems, lighting, sound, object motion, and behavioral simulation. In addition to these technical details, the book also addresses the creative and artistic issues involved so that you can design and generate more realistic and exciting environments. The package includes the latest version of the popular freeware VR renderer REND386, sample worlds and their code, the NorthCAD-3D computer-aided design (CAD) program, tons of utilities, and details on modifying the source code for adding new features. Please don't forget to rest on the seventh day.

Available now • 350 pages
ISBN: 1-878739-62-X
U.S. $26.95 Can. $37.95
Disk: 1–3.5″

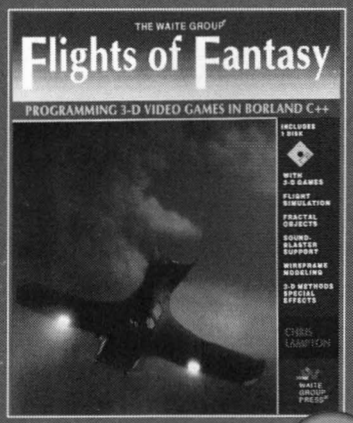


MODELING THE DREAM

Phil Shatz

More than 500 megabytes of the best computer animations on the planet, supported by exciting music, make for a package of truly mind-warping proportions. Take a dizzying ride through the insides of an arcade game, piloting an attack helicopter through an urban landscape, and bizarre androids fighting to the death. You'll also go behind the scenes to share the secrets of these talented artists and programmers.

November 1994 • 150 pages, ISBN: 1-878739-67-0, US. $32.95 Can. $45.95, Disk: 1–CD-ROM



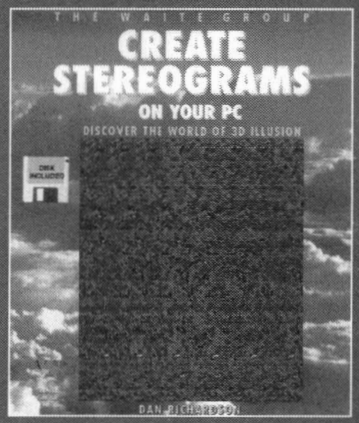
FLIGHTS OF FANTASY

BEST SELLER

Christopher Lampton

Take an in-depth, behind-the-scenes tour of writing 3-D video games with Borland C++. Learn to build a complete, full-featured flight simulator, modeled after the one included on the disk.

Available now • 560 pages
ISBN: 1-878739-18-2
U.S. $34.95 Can. $48.95
Disks: 1–3.5″

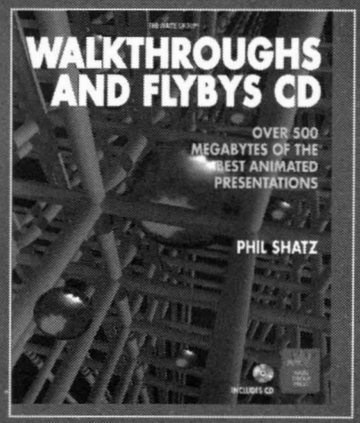


CREATE STEREOGRAMS ON YOUR PC

Dan Richardson

Stereograms go beyond flat 2D art into a world of astonishing, apparently random specks that suddenly collide in your vision to create 3D hologram-like illusions. This book/disk package provides step-by-step instructions and all the graphics tools needed to create your own stereograms on any PC.

Available now • 200 pages
ISBN: 1-878739-75-1
U.S. $26.95 Can. $37.95
(1 - 3.5″ disk)



WALKTHROUGHS AND FLYBYS CD

Phil Shatz

This book/CD package serves up awe-inspiring animated computer demos, complete with sound and music, and a behind-the-scenes look at their creation.

Available now • 150 pages
ISBN: 1-878739-40-9
U.S. $32.95 Can. $45.95
(1 - CD-ROM)

WAITE GROUP PRESS™

TO ORDER TOLL FREE CALL 1-800-368-9369

TELEPHONE 415-924-2575 • FAX 415-924-2576

OR SEND ORDER FORM TO: WAITE GROUP PRESS, 200 TAMAL PLAZA, CORTE MADERA, CA 94925

Qty	Book	US/Can Price	Total
___	Black Art of Windows Game Programming	$34.95/48.95	_____
___	Create Stereograms on Your PC	$26.95/37.95	_____
___	Fatal Distractions	$26.95/37.95	_____
___	Flights of Fantasy	$34.95/48.95	_____
___	Modeling the Dream CD	$32.95/45.95	_____
___	Playing God	$26.95/37.95	_____
___	Walkthroughs and Flybys CD	$32.95/45.95	_____
	Calif. residents add 7.25% Sales Tax		_____
	Shipping		
	USPS ($5 first book/$1 each add'l)		_____
	UPS Two Day ($10/$2)		_____
	Canada ($10/$4)		_____
	International ($30/book)		_____
	TOTAL		_____

Ship to

Name _____

Company _____

Address _____

City, State, Zip _____

Phone _____

Payment Method

☐ Check Enclosed ☐ VISA ☐ MasterCard

Card# _____ Exp. Date _____

Signature _____

SATISFACTION GUARANTEED OR YOUR MONEY BACK.

This is a legal agreement between you, the end user and purchaser, and The Waite Group®, Inc., and the authors of the programs contained in the disk. By opening the sealed disk package, you are agreeing to be bound by the terms of this Agreement. If you do not agree with the terms of this Agreement, promptly return the unopened disk package and the accompanying items (including the related book and other written material) to the place you obtained them for a refund.

SOFTWARE LICENSE

1. The Waite Group, Inc. grants you the right to use one copy of the enclosed software programs (the programs) on a single computer system (whether a single CPU, part of a licensed network, or a terminal connected to a single CPU). Each concurrent user of the program must have exclusive use of the related Waite Group, Inc. written materials.

2. The program, including the copyrights in each program, is owned by the respective author and the copyright in the entire work is owned by The Waite Group, Inc. and they are therefore protected under the copyright laws of the United States and other nations, under international treaties. You may make only one copy of the disk containing the programs exclusively for backup or archival purposes, or you may transfer the programs to one hard disk drive, using the original for backup or archival purposes. You may make no other copies of the programs, and you may make no copies of all or any part of the related Waite Group, Inc. written materials.

3. You may not rent or lease the programs, but you may transfer ownership of the programs and related written materials (including any and all updates and earlier versions) if you keep no copies of either, and if you make sure the transferee agrees to the terms of this license.

4. You may not decompile, reverse engineer, disassemble, copy, create a derivative work, or otherwise use the programs except as stated in this Agreement.

GOVERNING LAW

This Agreement is governed by the laws of the State of California.

LIMITED WARRANTY

The following warranties shall be effective for 90 days from the date of purchase: (i) The Waite Group, Inc. warrants the enclosed disk to be free of defects in materials and workmanship under normal use; and (ii) The Waite Group, Inc. warrants that the programs, unless modified by the purchaser, will substantially perform the functions described in the documentation provided by The Waite Group, Inc. when operated on the designated hardware and operating system. The Waite Group, Inc. does not warrant that the programs will meet purchaser's requirements or that operation of a program will be uninterrupted or error-free. The program warranty does not cover any program that has been altered or changed in any way by anyone other than The Waite Group, Inc. The Waite Group, Inc. is not responsible for problems caused by changes in the operating characteristics of computer hardware or computer operating systems that are made after the release of the programs, nor for problems in the interaction of the programs with each other or other software.

THESE WARRANTIES ARE EXCLUSIVE AND IN LIEU OF ALL OTHER WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR OF ANY OTHER WARRANTY, WHETHER EXPRESS OR IMPLIED.

EXCLUSIVE REMEDY

The Waite Group, Inc. will replace any defective disk without charge if the defective disk is returned to The Waite Group, Inc. within 90 days from date of purchase.

This is Purchaser's sole and exclusive remedy for any breach of warranty or claim for contract, tort, or damages.

LIMITATION OF LIABILITY

THE WAITE GROUP, INC. AND THE AUTHORS OF THE PROGRAMS SHALL NOT IN ANY CASE BE LIABLE FOR SPECIAL, INCIDENTAL, CONSEQUENTIAL, INDIRECT, OR OTHER SIMILAR DAMAGES ARISING FROM ANY BREACH OF THESE WARRANTIES EVEN IF THE WAITE GROUP, INC. OR ITS AGENT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THE LIABILITY FOR DAMAGES OF THE WAITE GROUP, INC. AND THE AUTHORS OF THE PROGRAMS UNDER THIS AGREEMENT SHALL IN NO EVENT EXCEED THE PURCHASE PRICE PAID.

COMPLETE AGREEMENT

This Agreement constitutes the complete agreement between The Waite Group, Inc. and the authors of the programs, and you, the purchaser.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above exclusions or limitations may not apply to you. This limited warranty gives you specific legal rights; you may have others, which vary from state to state.

SATISFACTION REPORT CARD

Please fill out this card if you wish to know of future updates to *Gardens of Imagination*, or to receive our catalog.

SATISFACTION CARD

Company Name: _____

Division/Department: _____ **Mail Stop:** _____

Last Name: _____ **First Name:** _____ **Middle Initial:** _____

Street Address: _____

City: _____ **State:** _____ **Zip:** _____

Daytime telephone: ( ) _____

Date product was acquired: **Month** _____ **Day** _____ **Year** _____ **Your Occupation:** _____

Overall, how would you rate *Gardens of Imagination*?

☐ Excellent ☐ Very Good ☐ Good
☐ Fair ☐ Below Average ☐ Poor

What did you like MOST about this book? _____

_____

What did you like LEAST about this book? _____

_____

_____

Please describe any problems you may have encountered with installing or using the disk: _____

_____

How did you use this book (problem-solver, tutorial, reference...)?

_____

What is your level of computer expertise?

☐ New ☐ Dabbler ☐ Hacker
☐ Power User ☐ Programmer ☐ Experienced Professional

What computer languages are you familiar with? _____

_____

Please describe your computer hardware:

Computer _____ Hard disk _____
5.25" disk drives _____ 3.5" disk drives _____
Video card _____ Monitor _____
Printer _____ Peripherals _____
Sound Board _____ CD ROM _____

Where did you buy this book?

☐ Bookstore (name): _____
☐ Discount store (name): _____
☐ Computer store (name): _____
☐ Catalog (name): _____
☐ Direct from WGP ☐ Other _____

What price did you pay for this book? _____

What influenced your purchase of this book?

☐ Recommendation ☐ Advertisement
☐ Magazine review ☐ Store display
☐ Mailing ☐ Book's format
☐ Reputation of Waite Group Press ☐ Other

How many computer books do you buy each year? _____

How many other Waite Group books do you own? _____

What is your favorite Waite Group book? _____

_____

Is there any program or subject you would like to see Waite Group Press cover in a similar approach? _____

_____

Additional comments? _____

_____

Please send to: **Waite Group Press**
Attn: *Gardens of Imagination*
200 Tamal Plaza
Corte Madera, CA 94925

☐ **Check here for a free Waite Group catalog**

THE WAITE GROUP®

GARDENS OF IMAGINATION

Programming 3D Maze Games in C/C++

You've outwitted and outrun the Nazi prison guards in Wolfenstein 3D and managed to escape mortal devastation in DOOM. But you couldn't even begin to fathom how the creators of these wildly popular computer maze games programmed these spitfire-paced, smooth-animated 3D masterpieces. Until now—*Gardens of Imagination: Programming 3D Maze Games in C/C++* blows the lid off the maze game programming mystery. The sequel to *Flights of Fantasy*, author Christopher Lampton's ground-breaking flight simulator programming guide, this intermediate book and accompanying disk give you all the technical know-how needed to write your own heart-pounding, spine-tingling maze games using Borland C++.

Gardens of Imagination clearly explains the best-kept maze programming secrets used in state-of-the-art games like Wolfenstein 3D. Lampton offers step-by-step instructions for such functions as maze design, texture mapping, and the hottest new ray casting techniques. You'll see how to cast variable light on inside walls, create stairs and table surfaces, and bitmap floors and ceilings. Plus you'll learn all the essential nuts and bolts, from programming standard input devices like the mouse and joystick to generating faster graphics using little-known VGA Mode X.

Additional features:

- Includes high-speed maze game and source code
- Support available from author on CompuServe GAMERS forum
- Design mazes and import graphics easily with the accompanying utility programs

Operating System: MS-DOS
Level: Intermediate/Advanced
Requires: 386 with a 387 or 486 DX , 4 MB RAM, 10 MB hard disk space, VGA (or SVGA) graphics board, Mouse, Borland C++ 3.1 or better

WAITE GROUP PRESS™
200 Tamal Plaza
Corte Madera, CA 94925

$34.95 USA
$48.95 Canada

Shelving: Computer Programming/Games

ISBN 1-878739-59-X
53495
EAN
9 781878 739599